PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Un algorithme tabou stochastique pour le problème de recouvrement d'ensemble à coûts unitaires
Auteur: Author:	Mohamed Wassim Bouzidi
Date:	2015
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Bouzidi, M. W. (2015). Un algorithme tabou stochastique pour le problème de recouvrement d'ensemble à coûts unitaires [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2013/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/2013/
Directeurs de recherche: Advisors:	Philippe Galinier
Programme: Program:	Génie informatique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

UNIVERSITÉ DE MONTRÉAL

UN ALGORITHME TABOU STOCHASTIQUE POUR LE PROBLÈME DE RECOUVREMENT D'ENSEMBLE À COÛTS UNITAIRES

MOHAMED WASSIM BOUZIDI
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2015

© Mohamed Wassim Bouzidi, 2015.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

UN ALGORITHME TABOU STOCHASTIQUE POUR LE PROBLÈME DE RECOUVREMENT D'ENSEMBLE À COÛTS UNITAIRES

présenté par : BOUZIDI Mohamed Wassim
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. GUIBAULT François, Ph. D., président
M. GALINIER Philippe, Doctorat., membre et directeur de recherche
M. GAGNON Michel, Ph. D., membre

DÉDICACE

À mes parents...

REMERCIEMENTS

Merci à mon directeur de recherche, Philippe Galinier, pour son grand intérêt pour mes travaux et son suivi régulier.
Merci à ma famille et mes amis pour leur soutien.

RÉSUMÉ

Le problème de recouvrement d'ensemble avec coûts unitaires (USCP) est un problème NP-difficile. Ce problème possède plusieurs applications importantes comme le problème d'affectation des équipages. Le but de notre travail est de résoudre de manière efficace le problème USCP. Pour atteindre cet objectif, nous avons commencé par développer un algorithme tabou qui s'inspire du meilleur algorithme conçu pour résoudre ce problème. L'un des points faibles de ce dernier algorithme est l'absence d'une technique permettant un réglage efficace des paramètres. Notre principal objectif était de trouver une manière efficace de régler les paramètres. Durant notre travail, nous avons exploré plusieurs approches. La première approche consistait à trouver des formules générales pour nos listes taboues. Nous n'avons pas réussi à trouver des formules simples, mais les résultats des tests réalisés avec nos deux formules compliquées sont meilleurs que ceux obtenus par le meilleur algorithme de la littérature. La deuxième approche consistait à adapter l'algorithme tabou réactif à notre problème USCP. Les tests réalisés avec cette approche ont montré que l'algorithme ne produit pas des cycles avec les jeux de grande taille, donc incapable de régler dynamiquement les longueurs des listes taboues. Notre troisième idée consistait à combiner le recuit simulé avec l'algorithme tabou. Nos tests ont révélé que l'algorithme obtient des résultats médiocres lorsque la température n'est pas suffisamment basse. Grâce aux résultats obtenus avec la troisième approche, nous avons développé notre algorithme tabou stochastique STS. Notre algorithme STS nous a permis de régler plus facilement les longueurs des listes taboues. Les résultats de STS sont meilleurs que ceux obtenus par RWLS – le meilleur algorithme de la littérature publié récemment. Notre algorithme obtient 6 nouveaux records et atteint tous les meilleurs résultats sur le reste des jeux de données. Pour rendre nos algorithmes plus rapides, nous avons développé une implémentation efficace. Notre implémentation est fondée sur deux caractéristiques clés. La première est l'utilisation d'algorithmes de bas niveau incrémentaux. Le deuxième point fort de notre implémentation est l'utilisation des files de priorité qui rendent la sélection d'un mouvement plus rapide. Les tests effectués montrent l'efficacité de nos files de priorités sur la majorité des jeux de données traités dans notre travail.

ABSTRACT

The unicost set covering problem (USCP) is an NP-hard problem. This problem has many important real-life applications such as the crew scheduling problem. In this work, we aim to effectively solve the USCP. To achieve this goal, we first developed a tabu search algorithm inspired by the best algorithm designed to solve the USCP. One of the weaknesses of the latter algorithm is the absence of an effective technique for setting the parameters. Our main objective was to find an effective way to adjust the tabu lists parameters. During our work, we explored several approaches. The first approach was to find general formulas for our tabu lists. We have not managed to find simple formulas, but the results of the tests performed with our two complicated formulas are better than those obtained by the best performing algorithms in the literature. The second approach was to adapt the reactive tabu algorithm to our problem. Tests performed with this approach have shown that the algorithm does not produce cycles when applied to big instances, so it is unable to dynamically adjust the length of the tabu lists. Our third idea was to combine a simulated annealing algorithm with the tabu algorithm. Our tests revealed that the algorithm performs poorly when the temperature is not low enough. Thanks to the results obtained with the third approach, we have developed our stochastic tabu algorithm STS. Our STS algorithm allowed us to easily adjust the lengths of the tabu lists. STS results are better than those obtained by RWLS - the best algorithm in the literature which was recently published. Our algorithm obtains 6 new records and achieves the best results on all the remaining instances. To make our algorithm faster, we have developed an efficient implementation. Our implementation is based on two features. The first is the use of incremental low level algorithms. The second feature of our implementation is the use of priority queues that make the selection of the movements faster. The tests show the effectiveness of using the priority queues on the majority of the instances used in this work.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

RWLS	Row Weighting Local Search
SATS	Simulated Annealing Tabu Search
SCP	Set Covering Problem
STS	Stochastic Tabu Search
TS	Tabu Search
USCP	Unicost Set Covering Problem

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Le problème de recouvrement d'ensemble (Set Covering Problem (SCP)) est l'un des problèmes classiques de l'optimisation combinatoire. Un cas particulier du problème SCP est constitué par le problème de *recouvrement d'ensemble avec coûts unitaires* (Unicost Set covering Problem (USCP)). Ces deux problèmes NP-difficiles possèdent des applications importantes dans différents domaines, notamment les problèmes de transport en commun et le problème d'affectation des équipages (voir Balas, 1982; Ceria et al., 1998). Le travail de recherche présenté dans ce mémoire vise à concevoir, développer et tester des *heuristiques* (algorithmes approchés) efficaces pour traiter le problème USCP. Dans ce chapitre, nous présentons successivement des définitions et des concepts de base, la problématique, les objectifs de notre recherche et le plan du mémoire.

1.1 Définitions et concepts de base

Définition des problèmes SCP et USCP Considérons un ensemble $M = \{1, \ldots, m\}$ et une famille $\mathcal{F} = \{M_1, \ldots, M_n\}$ de sous-ensembles de M dont la réunion couvre M : $M_1, \ldots, M_n \subseteq M$, $\bigcup_{i=1..n} M_i = M$. Nous définissons la densité d par : $d = \frac{1}{n \times m} \sum_{i=1}^{n} |M_i|$. De plus, soit $c : \{1..n\} \rightarrow \mathbb{R}^+$ une fonction qui associe un coût positif $c(i)$ à chacun des sous-ensembles M_i. Étant donné un sous-ensemble S de $\{1..n\}$, on dit que S est un recouvrement de M si et seulement si $\bigcup_{i \in S} M_i = M$. De plus, le coût de S est défini par $\sum_{i \in S} c(i)$.
Le problème de recouvrement d'ensemble consiste à trouver un recouvrement de coût minimum. Nous appelons cette version du problème la version pondérée. Le problème USCP correspond au cas où tous les sous-ensembles ont un coût égal à 1. Ce problème revient donc à trouver un recouvrement de cardinalité minimum.

Exemple introductif du problème USCP Soient l'ensemble $M = \{1, 2, 3, 4, 5, 6, 7\}$ et la famille $\mathcal{F} = \{M_1 = \{1, 3, 5\}, M_2 = \{\{2, 7\}, M_3 = \{\{4, 6, 3\}, M_4 = \{\{1, 5, 3, 6\}, M_5 = \{\{4, 2, 7\}\}$ de 5 sous-ensembles de M. Un recouvrement possible de M est $S = \{1, 2, 3\}$ car ces 3 sous-ensembles couvrent tous les éléments de M. Le recouvrement S qui utilise le moins de sous-ensemble est $S = \{4, 5\}$.

Problème NP-difficile et heuristique Selon la théorie de la complexité des problèmes, les deux problèmes SCP et USCP sont classés NP-difficiles. Selon cette théorie, il n'existe

pas d'algorithme polynomial permettant de résoudre de manière optimale un problème NP-difficile – en supposant que P $\neq$ NP. Pour résoudre un tel problème (de manière optimale), on doit donc utiliser un algorithme exponentiel. Cependant, on risque alors d'être confronté à l'explosion du temps de calcul. C'est pourquoi, pour traiter les exemplaires de grande taille des problèmes d'optimisation NP-difficiles, les chercheurs ont développé des techniques algorithmiques qui ne garantissent pas l'optimalité, mais qui permettent en pratique de produire des solutions aussi bonnes que possible (aussi proches que possible de l'optimum). Ces techniques sont appelées des heuristiques. De plus, on appelle *métaheuristique* un schéma d'heuristique générique (qui peut être adapté à différents problèmes). Dans ce mémoire, nous allons nous intéresser à une famille de métaheuristiques que nous appelons la recherche locale. La principale caractéristique de la recherche locale est d'exploiter une *fonction de voisinage* définie entre les solutions du problème à résoudre.

Une heuristique de recherche locale explore un ensemble d'états que nous appelons des *configurations*. L'ensemble des configurations est appelé l'*espace de recherche*. De plus, on définit une fonction de voisinage entre les configurations (voir le sous-chapitre 2.2.2). Un algorithme de recherche locale typique commence par construire une configuration initiale S_0 (par exemple simplement de manière aléatoire), puis engendre une série de configurations S_1, S_2, etc., en choisissant chaque nouvelle configuration S_{i+1} dans le voisinage de la précédente (S_i). À chaque itération i, le choix de la configuration S_i est guidé par une fonction de coût. L'idée consiste à faire en sorte que le coût de la configuration courante diminue progressivement afin de découvrir une configuration de coût aussi bas que possible. Pour résoudre un problème particulier à l'aide de la recherche locale, il faut donc :

1. Définir une *approche de recherche locale*, c'est-à-dire un triplet constitué de l'espace de recherche, de la fonction de coût et de la fonction de voisinage (définissant les mouvements).
2. Définir le critère de sélection du mouvement à effectuer lors de chaque itération (mécanisme de visite), ce qui revient à choisir une métaheuristique de recherche locale.

Les heuristiques de descente représentent la forme la plus simple de métaheuristique de recherche locale. Dans une telle heuristique, on choisit à chaque itération un voisin dont le coût est strictement inférieur à celui de la configuration courante. Malheureusement, une telle heuristique se trouve en général rapidement bloquée dans un optimum local de qualité médiocre. Les métaheuristiques de recherche locale dites évoluées utilisent des mécanismes conçus pour ressortir des optima locaux. La métaheuristique de recherche locale évoluée la plus ancienne (proposée au début des années 1980) est le recuit simulé (voir Kirkpatrick and Vecchi, 1983; Cerny, 1985). Le principe du recuit simulé consiste à utiliser des perturbations aléatoires pour échapper aux optima locaux. Ainsi, alors que la descente n'accepte

aucun mouvement qui dégrade le coût de la configuration, le recuit simulé accepte de manière contrôlée certaines dégradations occasionnelles. Ce mécanisme est contrôlé par un paramètre appelé la température. Dans le recuit simulé standard, le paramètre de température décroît progressivement au cours de la recherche. Une autre métaheuristique proposée peu après le recuit simulé est l'algorithme tabou (voir Glover, 1989, 1990; Glover and Laguna, 1997). Cet algorithme se caractérise par l'utilisation d'une structure appelée liste taboue qui permet à l'algorithme d'interdire certains mouvements (qui deviennent ainsi tabous). Ces interdictions ont pour effet d'empêcher l'algorithme de retourner vers les configurations visitées précédemment, ce qui permet efficacement d'échapper aux optima locaux. Le mécanisme tabou est contrôlé par un paramètre (la longueur de la liste taboue). L'algorithme tabou s'est souvent montré sensiblement plus efficace que le recuit simulé et il est considéré comme l'une des métaheuristiques parmi les plus efficaces.

La plupart des heuristiques possèdent un ou plusieurs paramètres, et la tâche consistant à déterminer leur valeur idéale est généralement complexe. En effet, la valeur idéale des paramètres peut varier fortement d'un exemplaire du problème à l'autre. Traditionnellement, on peut observer dans la littérature que la manière utilisée en pratique pour régler les paramètres d'une heuristique n'obéit pas à des principes rigoureux et bien définis. Assez souvent, les chercheurs réalisent des tests préliminaires en utilisant quelques exemplaires durant lesquels ils tentent de déterminer une formule qui exprime la valeur des paramètres en fonction des caractéristiques décrivant l'exemplaire. Certains auteurs ont tenté de proposer des techniques adaptatives génériques permettant de régler automatiquement les paramètres au cours de l'exécution d'une heuristique. Dans le cas de l'algorithme tabou, une telle technique, appelée *liste taboue réactive*, a été proposée par (Battiti and Tecchiolli, 1994).

Heuristiques proposées pour le problème USCP Les problèmes SCP et USCP ont été prouvés NP-difficile (voir Garey and Johnson, 1979). De nombreuses heuristiques ont été proposées dans la littérature pour traiter ces problèmes. Parmi les heuristiques proposées pour le problème SCP, nous pouvons citer (Beasley, 1990a; Caprara et al., 1999; Lan et al., 2007). Parmi les heuristiques proposées spécifiquement pour USCP, nous pouvons citer (Musliu, 2006; Bautista and Pereira, 2007; Naji-Azimi et al., 2010; Gao et al., 2015). Comme le problème USCP est un cas particulier du problème SCP, les heuristiques dédiées au problème SCP peuvent également lui être appliquées. Au moment où nous avons commencé notre étude et jusqu'à récemment, l'heuristique la plus efficace pour le problème USCP était, et de loin, l'algorithme tabou proposé par (Musliu, 2006). Très récemment, un nouvel algorithme tabou largement inspiré par celui de Musliu a été proposé par (Gao et al., 2015).

Un point fort de l'algorithme de Musliu est sa "stratégie de recherche locale", c'est-à-dire la

manière de définir l'espace de recherche, la fonction de coût et le voisinage. Le mécanisme tabou qu'il utilise peut être qualifié de standard pour ce genre de problème et il semble bien adapté. Une caractéristique de ce mécanisme est d'utiliser deux listes taboues (voir le sous-chapitre 2.3.3).

Un point faible de l'algorithme est la manière de régler les deux paramètres contrôlant le mécanisme tabou. En fait, l'auteur ne propose aucune technique bien définie. Dans ses tests, pour traiter un exemplaire particulier, il réalise des exécutions avec cinq combinaisons des paramètres. Cette technique présente deux inconvénients. Tout d'abord, cela nécessite de réaliser cinq exécutions et augmente d'autant le temps de calcul total. De plus, dans de nombreux cas, les cinq combinaisons de valeurs se trouvent mal adaptées et l'algorithme n'atteint pas les meilleurs résultats possible. Malgré ce point faible, cet algorithme a fait sensiblement mieux que ses concurrents. Par ailleurs, on peut observer que l'auteur ne décrit pas dans le détail les structures de données qu'il utilise.

L'algorithme (Row Weighting Local Search (RWLS)) proposé récemment dans (Gao et al., 2015) s'inspire des points forts de l'algorithme de Musliu, en introduisant un ensemble de modifications. En particulier, il utilise un mécanisme de pondération qui semble avoir un effet de diversification à long terme. De plus, RWLS utilise une technique automatique permettant de régler les paramètres de l'algorithme. Cet algorithme est plus efficace que celui de Musliu. Il a permis d'établir de nouveaux records pour plusieurs jeux de données.

1.2 Éléments de la problématique

D'un point de vue très général, l'objectif de notre étude consiste à concevoir, implanter et tester des heuristiques efficaces pour le problème USCP. Plus spécifiquement, nous sommes partis de l'algorithme tabou de Musliu dans lequel nous avons conservé les points qui nous semblaient forts : la stratégie de recherche locale et le mécanisme tabou. Dans cet algorithme Tabu Search (TS), nous avons visé à améliorer deux caractéristiques. La première est la conception d'une technique permettant de régler efficacement les paramètres de l'algorithme. La seconde est la mise au point d'une implémentation efficace destinée à rendre l'algorithme aussi rapide que possible.

Le réglage des paramètres de l'algorithme Notre algorithme tabou initial possède deux listes taboues et nécessite donc le réglage de deux paramètres. Ceci a rendu la question du réglage des paramètres d'autant plus complexe. Nous avons ainsi testé un ensemble de techniques en commençant par celles qui semblaient les plus évidentes, mais sans trouver une méthode miracle donnant des résultats parfaitement satisfaisants. Nous avons ainsi exploré

quatre approches suivantes.

La première approche envisagée a consisté à chercher une formule générale exprimant les deux paramètres en fonction des caractéristiques du problème (le nombre m d'éléments et n de sous-ensembles, ainsi que la densité d) et la taille de la solution.

La deuxième approche envisagée a été d'implanter une liste taboue réactive inspirée de celle proposée dans la littérature.

Une troisième approche a été d'hybrider l'algorithme tabou avec le recuit simulé. Nous avons donc implanté un algorithme, nommé Simulated Annealing Tabu Search (SATS), qui utilise des listes taboues et qui sélectionne à chaque itération un mouvement non tabou selon les probabilités définies par le recuit simulé. Comme l'algorithme de recuit simulé standard, l'algorithme SATS fait décroître progressivement le paramètre de température. Au cours de l'exécution de l'algorithme, le paramètre de température utilise tout un continuum de températures, en partant des plus élevées et en allant jusqu'aux plus basses. Ceci nous garantit donc que, à un moment ou à un autre de l'exécution, l'algorithme va utiliser des valeurs efficaces. Pour cette raison, il semble logique de penser qu'un tel algorithme serait peu sensible à la valeur des différents paramètres.

En nous basant sur les tests réalisés avec SATS, nous avons mis au point un algorithme tabou stochastique nommé Stochastic Tabu Search (STS) qui peut être vu comme une variante de SATS dans laquelle la température ne décroît pas, mais varie périodiquement entre un ensemble de valeurs très basses. Les paramètres de cet algorithme se sont révélés beaucoup plus faciles à régler que ceux de l'algorithme TS de départ. Les tests réalisés montrent que notre algorithme STS obtient des résultats satisfaisants sur les jeux de données standards de la littérature.

Implémentation de l'algorithme Un algorithme de recherche locale doit, à chaque itération, réaliser deux tâches : évaluer différents mouvements applicables à la configuration courante et sélectionner l'un de ces mouvements. Ces deux tâches (tout particulièrement la première) consomment en général la plus grande partie du temps de calcul. Dans notre étude, nous avons mis l'accent sur ces deux points. Tout d'abord, nous avons conçu et mis au point des structures de données et algorithmes de bas niveau permettant de recalculer efficacement le score des différents mouvements. Ces algorithmes sont incrémentaux, ce qui signifie qu'ils mémorisent certaines informations afin d'éviter la répétition des mêmes calculs d'une itération à l'autre. De plus, nous utilisons des structures de données additionnelles qui accélèrent la sélection du mouvement. Ces structures de données peuvent être vues comme des files de priorité (ou une extension des files de priorité). Elles sont implantées à l'aide de *Buckets*.

1.3 Objectifs de recherche

Le but de notre étude consiste à concevoir, implanter et tester des heuristiques efficaces pour le problème USCP. Notre point de départ est l'heuristique la plus efficace de la littérature au moment du démarrage de nos travaux : l'algorithme tabou proposé par Musliu. Nous chercherons à améliorer les performances de cet algorithme en considérant les deux objectifs généraux suivants :

- mettre en œuvre une technique pour régler les paramètres de l'algorithme tabou ;
- développer une implémentation efficace de l'algorithme.

En rapport avec le premier objectif général cité, nous poursuivrons les objectifs suivants :

- déterminer empiriquement une formule générale pour le réglage des paramètres de l'algorithme tabou TS, puis à évaluer l'algorithme tabou lorsqu'il utilise ces paramètres ;
- implanter et tester un mécanisme de liste taboue réactive ;
- tester une heuristique qui combine le recuit simulé et tabou et chercher à comprendre l'influence du réglage des paramètres (longueurs des listes taboues et température) sur les performances de l'algorithme afin de déterminer un réglage approprié.

En rapport avec le second objectif général cité, nous poursuivrons les objectifs suivants :

- implanter des algorithmes de bas niveau permettant de calculer efficacement la performance des différents mouvements ;
- implanter une technique permettant de sélectionner rapidement un mouvement (files de priorité).

1.4 Plan du mémoire

Ce mémoire est composé de 6 chapitres et organisé comme suit :

Dans le chapitre 2, nous donnons plus d'informations sur le problème USCP. Nous discutons aussi dans ce chapitre les approches développées dans la littérature pour résoudre ce problème. Pour faciliter la compréhension des approches et de notre travail, nous présentons dans ce même chapitre quelques métaheuristiques.

Le chapitre 3 présente une description détaillée des techniques que nous avons développées tout au long de notre travail de recherche. Nous décrivons en détail nos algorithmes SATS et STS.

Dans le chapitre 4, nous décrivons les structures de données que nous avons conçues, ainsi que les algorithmes de bas niveau développés pour implanter notre algorithme SATS.

Dans le chapitre 5, nous présentons les détails des résultats que nous avons obtenus tout au long de notre travail. Nous détaillons les résultats des deux variantes de notre algorithme appliquées au problème USCP. Pour évaluer les performances de nos techniques, nous allons aussi comparer nos résultats avec ceux obtenus par les autres approches développées pour résoudre ce problème.

Le chapitre 6 est la conclusion du mémoire.

CHAPITRE 2 REVUE DE LITTÉRATURE

Dans la première partie de ce chapitre, nous citerons quelques applications de notre problème USCP. La méthode utilisée dans ce travail pour résoudre le problème USCP est une heuristique. C'est pour cette raison que toute une partie sera réservée à la présentation des heuristiques. Dans la troisième partie, nous discuterons quelques méthodes qui ont été développées pour résoudre notre problème.

2.1 Les applications du problème USCP

Les applications du problème USCP incluent les emplois de temps, l'affectation des équipages, les problèmes de transport en commun, les tests logiciels, les logiciels de détection de virus dans les ordinateurs, etc. Une revue de ces applications a été faite par (Ceria et al., 1998; Balas, 1982). L'importance de ce problème réside dans le fait que plusieurs problèmes de satisfaction de contraintes peuvent être facilement résolus en un temps polynomial en utilisant les bonnes décompositions. Pour plus de détails sur ce sujet, les articles (Bodlaender, 2005; Gottlob et al., 2000) peuvent être consultés.

2.2 Présentation des métaheuristiques

En optimisation combinatoire, une heuristique est une méthode approchée qui permet de trouver en temps polynomial une solution réalisable, pas obligatoirement optimale.

Un très grand nombre de problèmes d'optimisation combinatoire appartiennent à la classe des problèmes NP-difficiles. Pour un tel problème, il n'existe pas d'algorithme polynomial – si P $\neq$ NP. Pour résoudre un tel problème, il faut donc utiliser un algorithme exponentiel. Avec un tel algorithme, le temps de calcul risque d'exploser avec l'augmentation de la taille de l'exemplaire du problème. Les heuristiques présentent une bonne alternative pour ce genre de problèmes, car elles offrent un bon compromis entre la qualité de la solution et le temps de calcul. Les métaheuristiques constituent des heuristiques génériques qui ne sont pas dédiées à un problème particulier, mais qui peuvent s'adapter à de nombreux problèmes. Les métaheuristiques sont souvent classées en quatre familles :

- les algorithmes constructifs, qui génèrent des solutions à partir d'une solution initiale vide, en ajoutant petit à petit des éléments jusqu'à l'obtention d'une solution complète ;
- les algorithmes de recherche locale, qui démarrent avec une solution complète et essaient de l'améliorer en explorant l'espace de recherche ;

– les algorithmes évolutionnaires, qui s'inspirent de la théorie de l'évolution ;
– les algorithmes hybrides, qui combinent différentes approches et qui regroupent aussi d'autres métaheuristiques n'appartenant pas aux trois autres familles.

Dans ce qui suit, nous décrivons ces quatre familles d'heuristiques (ou approches) et nous poursuivons par la description de quelques métaheuristiques particulières comme le recuit simulé et la recherche taboue. Pour écrire ce sous-chapitre, nous avons utilisé l'article (Hao et al., 1999).

2.2.1 L'approche constructive

C'est l'approche la plus ancienne parmi toutes les approches de l'optimisation combinatoire, mais elle garde toujours une place très importante. Une méthode qui utilise cette approche construit petit à petit une solution de la forme $S = (V_1, V_2, ...V_n)$. Partant d'une solution partielle qui peut être vide $S_0 = ()$, elle cherche à la compléter. Pour cela, l'approche détermine à chaque étape i la variable V_i et l'ajoute à la solution partielle. Le critère d'arrêt d'une approche constructive est l'obtention d'une solution complète. Durant la construction d'une solution, une méthode de construction utilise des critères heuristiques pour choisir les variables à chaque étape. Le critère heuristique utilisé diffère d'une méthode à une autre. L'efficacité d'une approche constructive dépend de sa capacité à utiliser efficacement les informations du problème. Les méthodes gloutonnes sont les méthodes les plus connues parmi toutes les méthodes de construction. Une méthode gloutonne construit la solution itérativement sans remettre en cause des choix déjà faits. Parmi les méthodes gloutonnes connues, nous pouvons citer l'heuristique gloutonne pour le problème de coloriage introduite par (Brélaz, 1979).

2.2.2 L'approche de recherche locale

La recherche locale est une classe qui regroupe des méthodes heuristiques anciennes. Durant la recherche, les méthodes de recherche locale manipulent des configurations complètes. Ces méthodes utilisent le concept de l'amélioration itérative. Après la génération d'une solution initiale (par exemple de façon aléatoire), l'algorithme explore l'espace de recherche, en engendrant, à chaque itération i, une nouvelle configuration S_i choisie dans le voisinage de la configuration S_{i-1}. Le choix de la nouvelle configuration parmi toutes les configurations voisines est guidé par une fonction de coût. Le but de l'heuristique consiste à faire diminuer progressivement le coût de la configuration courante afin de découvrir une configuration de coût aussi bas que possible.

Bien qu'elle soit basée sur des principes très simples, la recherche locale constitue une approche très puissante et constitue un ingrédient présent dans la plupart des heuristiques efficaces. La famille des métaheuristiques de recherche locale comprend l'algorithme de descente, l'algorithme de recuit simulé, l'algorithme tabou, l'algorithme de recherche locale itérée (ILS), la recherche à voisinage variable (VNS) et beaucoup d'autres. Plusieurs de ces métaheuristiques sont présentées plus loin dans cette section.

Soit X l'ensemble des configurations, pour traiter un problème particulier avec une heuristique de recherche locale, il faut définir les deux éléments suivants :
- la définition du voisinage $N : X \to 2^X$;
- la procédure qui permet d'exploiter le voisinage : la procédure qui détermine comment la recherche passe d'une configuration à une autre.

2.2.3 L'approche évolutionnaire

Cette classe de métaheuristiques est inspirée du processus d'évolution naturelle. Ces algorithmes ont été adoptés à l'optimisation grâce au travail de (Holland, 1975), et les travaux de (Goldberg, 1989a,b) ont contribué à enrichir ces algorithmes. Un algorithme évolutif est typiquement composé de trois éléments essentiels :

- une population : ensemble de solutions du problème ;
- un mécanisme d'évaluation : évaluer le score de chaque individu dans la population ;
- un mécanisme d'évolution : composé d'opérateurs comme la mutation et le croisement qui permettent la construction d'une nouvelle génération.

En pratique, un algorithme évolutionnaire commence avec une population initiale généralement composée d'individus générés aléatoirement et répète le cycle suivant : (1) mesurer la qualité des solutions dans la population, (2) sélectionner des individus et (3) construire une nouvelle génération à partir des individus sélectionnés. Ce processus est répété un certain nombre de fois.

2.2.4 Les algorithmes hybrides

Certaines métaheuristiques tentent de combiner plusieurs approches afin de bénéficier de tous leurs avantages. Un mode d'hybridation très populaire consiste à combiner l'approche de recherche locale et l'approche évolutionnaire. L'idée consiste à combiner la puissance de la recherche locale appliquée sur chaque individu de la population, et le processus évolutif appliqué à toute la population. Un tel algorithme hybride est appelé un algorithme mémétique. L'algorithme mémétique applique dans une première étape un algorithme de voisinage

aux individus de la population pendant un certain nombre d'itérations et fait appel ensuite à un processus évolutif, en utilisant notamment un opérateur de croisement. Adapter les opérateurs de la recherche locale et du mécanisme d'évolution utilisés au problème est très important.

2.2.5 L'algorithme de descente

La descente est la méthode de recherche locale la plus élémentaire et la plus facile à implémenter. À chaque itération, la recherche locale essaye de trouver un voisin qui diminue la valeur de la fonction de coût. Plusieurs options sont possibles pour atteindre cet objectif :

– soit on parcourt les voisins de la solution courante jusqu'à ce qu'on trouve un voisin qui améliore la fonction de coût ;
– soit on passe en revue tous les voisins et on choisit le meilleur. Il est évident que cette possibilité est plus coûteuse, mais plus efficace.

L'algorithme de descente se bloque en général rapidement dans un optimum local de qualité médiocre. D'autres métaheuristiques de recherche locale dites évoluées comme le recuit simulé et la recherche taboue utilisent des mécanismes conçus pour ressortir des optima locaux.

2.2.6 L'algorithme du recuit simulé

Le recuit simulé (SA), qui a été conçu par (Kirkpatrick and Vecchi, 1983; Cerny, 1985), est la première vraie métaheuristique. La conception de cette métaheuristique s'inspire d'une technique, utilisée par les métallurgistes, nommée recuit physique. Pour obtenir un alliage de bonne qualité, les métallurgistes appliquent des cycles de refroidissement et de réchauffage au matériau pour minimiser l'énergie. Un état d'énergie minimal correspond à une structure stable du matériau. En partant d'une température très haute, le matériau se trouve dans un état liquide, et l'application d'une phase de refroidissement progressive le conduit à retrouver sa forme solide. Cette technique est le fruit des travaux de thermodynamique réalisés par (Metropolis et al., 1953). Par analogie, le principe du recuit simulé est de chercher itérativement des configurations de coût plus faible et d'accepter aussi, d'une manière contrôlée par l'utilisateur, de temps en temps des mouvements qui dégradent. À chaque itération, un voisin $s' \in N(s)$ est généré aléatoirement. Si la configuration s' est de performance supérieure ou égale à s alors le mouvement est accepté. Dans le cas contraire, c'est à dire $f(s') > f(s)$, le mouvement est accepté selon une portabilité P qui dépend de deux éléments :

– la dégradation de la fonction $\delta(f)$ – les dégradations les plus faibles sont plus faciles à accepter ;
– la température T qui est un paramètre de contrôle – plus la température est élevée plus

les dégradations sont facile à accepter.
La probabilité P est calculée suivant la distribution de Boltzmann :

$$P = e^{\delta(f)/T} \tag{2.1}$$

La température suit un schéma de refroidissement défini à travers deux paramètres : la longueur des paliers et la fonction décroissante des valeurs de la température. L'algorithme s'arrête en général soit lorsque la température atteint zéro ou lorsqu'aucune configuration voisine n'a été acceptée pendant un certain nombre d'essais.

Le schéma de refroidissement joue un rôle important dans l'efficacité du recuit simulé. Plusieurs techniques sont possibles :

1. réduction par paliers : on maintient la température durant un certain nombre d'itérations qui correspond au palier, et on la fait décroître après la fin du palier ;
2. réduction continue : la température décroît après chaque itération ;
3. réduction non monotone : la température décroît de façon continue, mais avec des augmentations occasionnelles.

2.2.7 L'algorithme tabou

La métaheuristique taboue est une méthode de voisinage, utilisant des techniques qui permettent d'éviter les optima locaux et les cycles. Cette métaheuristique a été développée par (Glover, 1989, 1990; Glover and Laguna, 1997) et elle connaît beaucoup de succès grâce aux résultats très satisfaisants obtenus sur un grand nombre de problèmes. L'objectif de cette métaheuristique est la diversification à court terme, en utilisant un mécanisme appelé liste taboue. Contrairement au recuit simulé, qui gère une configuration à chaque étape, la recherche taboue gère un ensemble de configurations appartenant à $N(s)$ et ne retient que le meilleur voisin s' même si $f(s') > f(s)$. Cette méthode ne se bloque pas dans un optimum local, mais elle peut entraîner des cycles courts. Le rôle de la liste taboue est d'empêcher ce genre de cycles en mémorisant les k dernières configurations et empêchant les mouvements qui conduisent à ces k dernières configurations. Le paramètre principal de cette métaheuristique est la taille de la liste taboue k, qui dépend du problème et peut varier au cours de la recherche. Le choix de la valeur de ce paramètre est très crucial parce qu'une liste trop longue peut être restrictive alors qu'une trop courte peut être inutile dans la majorité des cas. Mémoriser des configurations entières serait trop coûteux en termes de mémoire et temps de calcul. En pratique, la liste taboue mémorise des caractéristiques pour chacune des k configurations.

Lorsque la liste taboue mémorise des caractéristiques, les interdictions qu'elle engendre peuvent devenir trop fortes. Pour pallier ce problème, nous pouvons introduire un mécanisme d'aspiration. Ce mécanisme permet de lever le statut tabou d'un mouvement, si par exemple le mouvement permet d'atteindre une solution meilleure que la solution déjà trouvée jusqu'alors.

D'autres modifications peuvent être appliquées pour améliorer l'efficacité de la recherche taboue, comme l'intensification et la diversification. Ces deux techniques utilisent une mémoire à long terme. L'intensification utilise les propriétés communes des meilleures solutions rencontrées et favorise ces propriétés durant les phases d'intensification. La diversification peut être vue comme l'inverse de l'intensification. La diversification guide la recherche vers des régions de l'espace de recherche qui n'ont pas été visitées.

2.2.8 L'algorithme tabou réactif

Malgré l'efficacité de l'algorithme tabou, le réglage des paramètres qui contrôlent le mécanisme tabou pose toujours des problèmes. La clé pour développer un algorithme tabou efficace est de trouver les valeurs idéales des paramètres de la liste taboue. Si la liste taboue est trop courte, l'algorithme risque de tourner autour d'un optimum local, et, si par contre la liste est trop longue, la dose de diversification est trop forte et l'algorithme risque de diverger plutôt que converger. Pour résoudre ce casse-tête, Battiti et al proposent dans leurs articles (Battiti and Tecchiolli, 1994; Battii and Protasi, 2001) la recherche taboue réactive dans laquelle la longueur de la liste taboue s'adapte dynamiquement au cours de la recherche. La longueur diminue progressivement tant que l'algorithme ne cycle pas. Si l'algorithme cycle, une diversification est alors nécessaire ; la liste taboue s'adapte dynamiquement en s'allongeant.

2.2.9 L'algorithme GRASP

La métaheuristique GRASP (Greedy Randomized Adaptive Search Procedure) est une procédure itérative introduite par (Feo and Resende, 1989, 1995). La méthode GRASP combine les avantages des méthodes aléatoires et des méthodes gloutonnes. La métaheuristique GRASP répète les deux étapes suivantes :

1. Une étape de construction : une solution est construite itérativement. À chaque étape, on utilise une liste de sous-ensembles qu'on appelle liste de candidats construite de façon gloutonne et on choisit un élément de façon aléatoire de cette liste. On met à jour la liste des candidats après chaque itération. Cette étape se termine lorsqu'une solution complète est construite.

2. Une étape de descente : on applique une descente sur la solution complète construite dans la première étape.

Les deux étapes sont appliquées itérativement et retournent la meilleure solution trouvée. Les deux paramètres principaux de cette méthode sont la longueur de la liste des candidats et le nombre d'itérations

2.3 Les heuristiques proposées pour les problèmes SCP et USCP

Beaucoup de méthodes ont été développées pour résoudre le problème USCP. L'objectif de toutes ces méthodes est de trouver un recouvrement légal (une solution qui couvre tous les éléments) de cardinalité minimum. Dans un premier temps, nous allons présenter les méthodes mathématiques qui ont été développées pour résoudre le problème SCP. Nous allons aussi présenter les méthodes, basées sur des métaheuristiques, proposées pour résoudre les deux problèmes SCP et USCP.

2.3.1 Les méthodes mathématiques

Plusieurs heuristiques basées sur des relaxations lagrangiennes, des méthodes de sous-gradients et des branchements locaux ont été développées pour attaquer le problème SCP, comme les travaux de (Beasley, 1990a; Caprara et al., 1999; Yaghini et al., 2014). Ces méthodes sont efficaces sur la version pondérée du problème SCP. Malheureusement, la majorité des auteurs qui utilisent ce genre de méthodes ne testent pas leurs algorithmes sur les jeux de données USCP. Seuls les auteurs de l'article (Yaghini et al., 2014) ont testé leur heuristique sur des jeux de données USCP, en particulier sur les exemplaires de la famille *CYC*. Les auteurs de cette heuristique obtiennent les meilleurs résultats de la littérature sur les jeux de données SCP, mais leur heuristique ne fonctionne pas bien sur les jeux *CYC*. En particulier, leurs résultats sur ces exemplaires sont loin derrière les résultats obtenus dans (Musliu, 2006; Gao et al., 2015).

2.3.2 Heuristiques proposées pour le problème SCP

Dans la littérature, plusieurs heuristiques ont été conçues spécialement pour le problème SCP. Parmi les plus connues, nous pouvons citer l'heuristique Meta-RaPS qui a été proposée dans (Lan et al., 2007). L'heuristique Meta-RaPS commence par générer une solution réalisable en introduisant de l'aléatoire dans une méthode constructive développée par (Chvatal, 1979). Une fois une solution réalisable obtenue, tous les sous-ensembles redondants sont supprimés de cette solution. Durant la deuxième phase de l'algorithme, une heuristique de recherche locale

est appliquée pour améliorer la qualité de la solution obtenue durant la première phase. Cet algorithme atteint les meilleurs résultats sur les jeux de données SCP et obtient des résultats meilleurs que ceux des méthodes mathématiques sur les jeux de données USCP. Une autre heuristique basée sur les algorithmes génétiques a été proposée récemment par (Bilal, 2014). Dans son travail, Bilal développe un algorithme mémétique nommé GATS dans lequel un algorithme tabou et un algorithme génétique sont combinés. Son algorithme tabou utilise une nouvelle fonction de pénalité qui a été proposée dans (Bilal et al., 2013) grâce à une nouvelle formulation mathématique du problème SCP. Une autre nouveauté dans GATS est le contrôle des paramètres tabous par l'algorithme génétique. Sur les jeux de données USCP, l'algorithme GATS obtient d'aussi bons résultats que ceux obtenus par l'algorithme Meta-RaPS et l'algorithme de Musliu.

2.3.3 Un algorithme tabou

L'approche de recherche locale développée par (Musliu, 2006) était considérée comme le meilleur algorithme de recherche locale pour résoudre le problème USCP avant la publication de l'article (Gao et al., 2015). Dans son travail, Musliu a utilisé une stratégie de recherche locale efficace. Sa stratégie de recherche locale repose sur deux éléments clés :

– elle utilise le voisinage ADD/REM constitué de mouvements d'insertion et de suppression d'un sous-ensemble ;
– elle restreint l'espace de recherche : si une nouvelle solution S de taille k est trouvée, alors l'algorithme explore l'ensemble des solutions de taille inférieure ou égale à $k - 1$.

Un autre élément clé de l'algorithme est la fonction de coût utilisée. Cette fonction qui guide la recherche est définie comme le nombre d'éléments non couverts dans la solution, plus la cardinalité de la solution. L'algorithme de Musliu utilise un mécanisme tabou simple avec deux listes taboues. La première liste est pour les sous-ensembles dans la solution et l'autre liste est pour les sous-ensembles candidates pour entrer dans la solution. Si une solution est obtenue après l'ajout ou le retrait d'un sous-ensemble, alors ce dernier est ajouté à l'une des deux listes taboues et l'algorithme ne peut ni le retirer ni l'ajouter pour un certain nombre d'itérations. Voici ci-dessous les étapes de l'algorithme de Musliu :

1. générer la solution initiale ;
2. initialiser la liste taboue et le paramètre k ;
3. générer le voisinage de la solution courante (on ne considère que les mouvements d'ajout et de retrait) ;
4. évaluer les solutions du voisinage (la fonction d'évaluation définie ci-haut) ;

5. appliquer le meilleur mouvement ;
6. mettre à jour la liste taboue et la valeur du paramètre k ;
7. aller à l'étape 3 si le critère d'arrêt n'est pas satisfait sinon aller à l'étape 8 ;
8. retourner la meilleure solution trouvée.

2.3.4 Un algorithme GRASP

Bautista et al. ont développé un algorithme glouton randomisé pour la résolution du problème USCP (voir Bautista and Pereira, 2007). Cet algorithme est composé de deux phases :

– Dans la première phase, l'algorithme commence par une solution vide. À chaque étape de cette phase constructive, une liste restreinte de candidats RLC est construite suivant une fonction de coût. Selon cette fonction, le score d'un sous-ensemble correspond au nombre d'éléments qu'il va couvrir si on l'insère dans la solution. L'algorithme choisit un sous-ensemble aléatoirement parmi tous les candidats dans la RLC. Cette phase se termine lorsqu'une solution légale est trouvée.
– Dans la phase de recherche locale, les auteurs ont choisi d'utiliser un voisinage basé sur des mouvements d'échanges. Un mouvement d'échange consiste à retirer un sous-ensemble de la solution courante et d'insérer un sous-ensemble qui n'appartient pas à la solution. À chaque itération, la procédure choisit selon une probabilité p de faire soit le meilleur mouvement d'échange possible soit un mouvement d'échange aléatoire pour échapper aux optima locaux.

2.3.5 Une métaheuristique électromagnétique

Une nouvelle métaheuristique qualifiée d'électromagnétique a été proposée dans (Naji-Azimi et al., 2010) sur le problème USCP. La métaheuristique utilise les concepts de l'électromagnétisme (Birbil and Fang, 2003). Les solutions dans la population sont vues comme des particules chargées, et la charge de la solution dépend de la fonction objective utilisée dans l'algorithme. L'exploration du voisinage dépend de la charge de toutes les solutions. Dans leur travail, les auteurs commencent par la construction d'une population initiale avec une méthode gloutonne. Ensuite, la recherche locale est appliquée à toutes les solutions présentes dans la population. La théorie de l'électromagnétisme intervient après cette étape pour faire évoluer les solutions de la population dans l'espace de recherche, en générant des mouvements en fonction des valeurs de la fonction de coût. À la fin de l'algorithme, une dernière phase de mutation est appliquée à la population.

2.3.6 L'algorithme RWLS

Un algorithme tabou efficace a été développé récemment dans (Gao et al., 2015). Les auteurs de ce travail se sont inspirés des points forts de l'algorithme de (Musliu, 2006), en apportant des améliorations pour corriger les points faibles de ce dernier algorithme. L'algorithme RWLS utilise le même voisinage ADD/REM utilisé par Musliu. Pour les mouvements d'ajout, les auteurs implémentent la technique Min-Conflicts (voir Minton et al., 1992) : pour ajouter un sous-ensemble, RWLS choisit aléatoirement un élément non couvert et insère dans la solution le meilleur sous-ensemble admissible qui couvre cet élément. L'algorithme utilise un mécanisme de pondération des éléments. Ce mécanisme de pondération est défini dans leur travail comme un mécanisme de diversification. Pour les deux types de mouvements, l'algorithme utilise le même mécanisme tabou que Musliu. Contrairement au réglage tabou de Musliu, le réglage tabou de l'algorithme RWLS est un réglage automatique qui ne dépend pas de l'exemplaire traité.

CHAPITRE 3 ALGORITHMES PROPOSÉS

Dans ce chapitre, nous décrivons l'algorithme SATS (Simualted Annealing Tabu Search) que nous proposons pour le problème de recouvrement d'ensemble à coûts unitaires (USCP). Cet algorithme constitue un hybride du recuit simulé et de l'algorithme tabou. Ci-dessous, nous présentons les différentes caractéristiques de cet algorithme, en particulier l'approche de recherche locale, le mécanisme tabou et la sélection d'un mouvement. Finalement, nous détaillons le pseudo-code de l'algorithme SATS, ainsi que celui d'une variante de l'algorithme SATS que nous appelons l'algorithme STS (Stochastic Tabu Search).

3.1 L'approche de recherche locale

Dans ce sous-chapitre, nous décrivons l'approche de recherche locale utilisée, c'est-à-dire la manière dont nous définissons l'espace de recherche (configurations), la fonction de pénalité et la fonction de voisinage (mouvements).

Configurations et mouvements Dans l'algorithme SATS, une configuration S correspond à tout sous-ensemble de l'ensemble $N = \{1..n\}$ de variables : $S \subseteq N$. On notera qu'une configuration peut être réalisable (être un recouvrement légal) ou non (s'il reste des éléments non couverts par les sous-ensembles de S) (voir le sous-chapitre 1.1)

L'algorithme SATS peut effectuer deux types de mouvements : des mouvements d'insertion et des mouvements de suppression. Un mouvement d'insertion, noté $<+, y>$ consiste à insérer dans la configuration courante un nouveau sous-ensemble y. Un mouvement de suppression, noté $<-, x>$, consiste à retirer de la configuration courante un élément x. Étant donné une configuration S et un mouvement m quelconque applicable à S, on note $S \oplus m$ la configuration obtenue en appliquant le mouvement m à S. On a donc :
– pour tout $x \in S$, $S \oplus <-, x> = S \backslash \{x\}$;
– pour tout $y \notin S$, $S \oplus <+, y> = S \cup \{y\}$.
Lors d'une itération de l'algorithme SATS, des règles particulières s'appliquent pour déterminer si on doit effectuer un mouvement d'insertion ou de suppression. Ces règles seront précisées plus bas dans le sous-chapitre 3.4.

Fonction de pénalité Pour guider la recherche vers les solutions réalisables, l'algorithme SATS utilise une fonction de pénalité. Pour une configuration quelconque S, on note $U(S)$

l'ensemble de contraintes violées (éléments non couverts) par S :

$$U(S) = \{i \in M \mid i \notin M_j, \forall j \in S\} \tag{3.1}$$

On définit la pénalité $f(S)$ de la configuration S comme le nombre de contraintes violées (éléments non couverts) dans S :

$$f(S) = |U(S)| \tag{3.2}$$

Score d'un mouvement Dans la suite, on note S la configuration courante de l'algorithme SATS. Étant donné un mouvement quelconque m (d'insertion ou de suppression), on note $\delta(m)$ le score du mouvement m, défini comme l'impact du mouvement sur le coût de la configuration courante. On a donc :

$$\delta(m) = f(S \oplus m) - f(S) \tag{3.3}$$

3.2 Mécanisme tabou

Les principes généraux d'un algorithme tabou ont été présentés dans le sous-chapitre 2.2.7. On rappelle qu'un algorithme tabou utilise le mécanisme tabou afin d'échapper aux optima locaux. Ce mécanisme se fonde sur la notion de liste taboue.

Listes taboues Pour le problème USCP, nous utilisons un mécanisme tout à fait classique pour ce genre de problème ; ce mécanisme est identique à celui utilisé dans (Musliu, 2006). Le principe est le suivant. Après avoir réalisé un mouvement d'insertion, on interdit pendant quelques itérations à l'algorithme de retirer l'élément qui vient d'être inséré dans la configuration. De la même manière, un élément qui vient d'être supprimé ne pourra pas être réinséré durant quelques itérations.

Nous utilisons deux listes taboues nommées *TLin* et *TLout*. *TLin* contient des variables qui n'appartiennent pas à S (donc potentiellement entrantes) et *TLout* des variables qui appartiennent à S (potentiellement sortantes). Si une variable est contenue dans *TLin*, cette variable entrante est dite taboue et, tant qu'elle reste dans *TLin*, elle ne peut pas être introduite dans la solution. De la même manière, une variable de *TLout* ne peut pas être retirée de la solution. Après un mouvement d'insertion $<+, y>$, la variable y est insérée dans *TLout* pour *ttOut* itérations. Après un mouvement de suppression $<-, x>$, la variable x est insérée dans *TLin* pour *ttIn* itérations. Les valeurs de *ttOut* et *ttIn* sont des paramètres de l'algorithme. La manière de fixer la valeur de ces paramètres sera traitée au chapitre 5.

Critère d'aspiration Dans un algorithme tabou, un mouvement même tabou peut être néanmoins appliqué, mais à condition qu'il vérifie ce qu'on appelle un *critère d'aspiration*. Autrement dit, le critère d'aspiration permet de lever le statut tabou d'un mouvement.

Dans notre algorithme SATS, un mouvement tabou m vérifie le critère d'aspiration s'il permet d'atteindre une solution de pénalité nulle (c'est-à-dire un recouvrement légal), c'est-à-dire s'il vérifie l'équation :

$$\delta(m) = -f(S) \tag{3.4}$$

Mouvement candidat On appelle mouvement candidat un mouvement qui est soit non tabou, soit vérifie le critère d'aspiration. C'est parmi ces mouvements candidats que l'heuristique choisira à chaque itération le mouvement à appliquer.

3.3 Sélection biaisée d'un mouvement

Dans un algorithme tabou pur (contrairement à SATS, qui est un algorithme tabou avec le recuit simulé), on choisit à chaque itération un mouvement optimal (de score minimum) – en tirant au hasard parmi les ex æquo. Nous appelons cette technique le mode de choix TS. Dans l'algorithme SATS, l'hybridation avec le recuit simulé consiste précisément à permettre de ne pas choisir systématiquement l'un des mouvements candidats optimaux, mais aussi, selon une probabilité contrôlée, un mouvement candidat de score inférieur (mouvement sous-optimal).

Sélection d'un mouvement dans SATS Formellement, le critère de choix est le suivant : à chaque mouvement $m \in C$, où C désigne l'ensemble des mouvements candidats, on affecte un poids égal à $w(m) = \alpha^{\delta(m)}$, puis on choisit l'un des mouvements selon une probabilité proportionnelle à son poids – donc, selon la technique de la roulette biaisée. La probabilité de sélection $P(m)$ d'un mouvement m est donc définie comme suit :

$$P(m) = \alpha^{\delta(m)} / \sum_{i \in C} \alpha^{\delta(i)} \tag{3.5}$$

Le paramètre α est un réel choisi dans l'intervalle $[0, 1]$.

Coefficient de randomisation En analysant l'équation (3.5), on peut observer que, lorsque le coût d'un mouvement augmente d'une unité, sa probabilité de choix est multipliée par α. Plus la valeur de α est basse (proche de 0), plus le biais est fort en faveur des meilleurs mouvements. Ainsi, si $\alpha = 1$, tous les mouvements candidats auront une probabilité égale, ce qui correspond à un choix uniforme. Inversement, si $\alpha \to 0$, le choix s'apparente

au mode de choix TS décrit plus haut. Noter qu'on autorisera α à prendre n'importe quelle valeur entre 0 et 1, y compris 0 : dans le cas $\alpha = 0$, on réalisera un choix selon le mode TS (un choix uniforme entre les meilleurs mouvements).

Nous appellerons α le paramètre de randomisation. Noter que ce paramètre joue le même rôle que le paramètre de température dans le recuit simulé et qu'il suit un schéma de décroissance tout comme la température dans le recuit simulé. Cependant, alors que α est compris entre 0 et 1, la température T est comprise entre 0 et $+\infty$. Plus précisément α et T sont reliés par l'équation suivante :

$$\alpha = e^{-1/T} \tag{3.6}$$

Remarque On notera que la manière classique de choisir un mouvement dans le recuit simulé consiste à choisir répétitivement un mouvement aléatoire et à décider à l'aide du critère de Metropolis si ce mouvement sera appliqué ou s'il sera rejeté. La technique que nous avons présentée plus haut est différente puisqu'elle consiste à choisir "directement" un mouvement en fonction de probabilités définies pour les différents mouvements. Dans la littérature, cette technique s'appelle le recuit simulé sans rejet (*rejectionless annealing*) – (voir Greene and Supowit, 1986).

3.4 Pseudo-code de l'algorithme SATS

L'algorithme SATS prend en entrée les paramètres suivants : la configuration initiale S_0, le nombre maximum d'itérations *nbIter*, les deux paramètres *ttIn*, *ttOut* permettant de régler les listes taboues, la température initiale α_0 et le taux de décroissance β de la température.

Procédure *SATS*()

$S :=$ *buildInitSolution*()
TSin.init() ; *TSout.init*() ;
$\alpha := \alpha_0$
for *iter* = 1..*nbIter* **do**
    if $f(S) = 0$ **then**
        $S^* := S$
        kTarget := $|S^*|$ - 1
        typeOfMove := *removal*
    else if |S| = *kTarget* **then**

typeOfMove := *removal*
else
typeOfMove := *insertion*
if *typeOfMove* = *insertion* **then**
y := *chooseMoveInsertion*(α)
applyMoveInsertion(y)
tt := *randomize*(*ttOut*)
TSout.renderTabuUntil(y, *iter+tt*)
else
x := *chooseMoveRemoval*(α)
applyMoveRemoval(x)
tt := *randomize*(*ttIn*)
TSin.renderTabuUntil(x, *iter+tt*)
$\alpha := \alpha \times (1 - \beta)$
Return (S^*)

Au début, les listes taboues sont initialisées à vide et la température fixée à sa valeur initiale. Ensuite, l'algorithme réalise une série de *nbIter* itérations. À chaque itération, l'algorithme détermine en fonction de la taille de la configuration courante le type du mouvement à réaliser (insertion ou suppression). Puis, il sélectionne un mouvement candidat selon la procédure de sélection randomisée décrite plus haut au sous-chapitre 3.3, en utilisant la valeur courante de la variable α. Finalement, il insère le sommet impliqué dans le mouvement (inséré ou supprimé) dans la liste taboue appropriée. Voici ci-dessous quelques précisions supplémentaires.

Choix du type de mouvement L'algorithme utilise une variable nommée *kTarget*. Cette variable fixe la taille maximum de la configuration. Ainsi, si la taille de la configuration est égale à *kTarget*, l'algorithme doit effectuer un mouvement de suppression – puisqu'un mouvement d'insertion aurait pour effet de produire une configuration de taille *kTarget+1*. À tout instant, la valeur de la variable *kTarget* est fixée à la taille de la plus petite configuration légale trouvée, moins une unité. Elle est donc décrémentée lorsque l'algorithme trouve une configuration de coût nul.

Procédure d'initialisation Le rôle de la procédure *buildInitSolution*() est de construire la solution initiale utilisée par l'algorithme. Cette configuration est construire de manière semi-gloutonne. La procédure part d'un ensemble vide de variables $S = \{\}$. Puis, elle réalise un

ensemble d'insertions jusqu'à atteindre une configuration de coût nul (une solution réalisable). À chaque itération, trois sous-ensembles de $N - S$ sont tirés au hasard et la procédure choisit celui qui permet de recouvrir le plus grand nombre d'éléments.

Procédure de randomisation La procédure *randomize*(a) modifie aléatoirement la valeur de l'argument : elle renvoie aléatoirement un nombre compris entre $\frac{2\times a}{3}$ et $\frac{4\times a}{3}$. Elle est utilisée dans l'algorithme pour randomiser la longueur des paramètres tabous *ttIn* et *ttOut* ce qui a pour but de réduire le risque de cyclage de l'algorithme.

Paramètres de l'algorithme SATS Nous présentons les paramètres de l'algorithme SATS dans le tableau 3.1.

Tableau 3.1 Paramètres de l'algorithme SATS

paramètre	**description**
nbIter	nombre total d'itérations
α	coefficient de randomisation (analogue à la température)
β	taux de décroissance de la température à chaque itération
ttIn et *ttOut*	paramètres fixant la durée de rétention d'une variable dans les listes taboue

3.5 L'algorithme STS

Des tests réalisés avec l'algorithme SATS sont présentés au chapitre 5. Ces tests montrent que l'utilisation d'une température initiale trop élevée tend fortement à dégrader la qualité des résultats. Finalement, la meilleure version de SATS est obtenue en utilisant une température constante. Nous appelons cet algorithme STS, et nous le décrivons ci-dessous. L'algorithme STS que nous proposons est une variante de l'algorithme SATS. Dans cette variante, le schéma de décroissance de la température n'est pas utilisé. Dans l'algorithme STS, la température peut prendre trois valeurs différentes α_1, α_2, et α_3 ($\alpha_1 < \alpha_2 < \alpha_3$). La température α change périodiquement (toutes les *period* itérations) en suivant l'ordre $(\alpha_2, \alpha_1, \alpha_2, \alpha_3)$, puis en recommençant. Autrement dit, la valeur médiane alterne tantôt avec la plus petite valeur, tantôt avec la plus grande. Nous avons choisi ces valeurs de manière à ce qu'elles restent inférieures à 10^{-2}. Les valeurs choisies pour les paramètres sont *period* =1000, $\alpha_1 = 10^{-3}$, $\alpha_2 = 2.5 \times 10^{-3}$, et $\alpha_3 = 5 \times 10^{-3}$. La manière de fixer les deux paramètres fixant la longueur des listes taboues est décrite au sous-chapitre 5.3.1.

Les paramètres de l'algorithme STS sont présentés dans le tableau 3.2.

Tableau 3.2 Paramètres de l'algorithme STS

paramètre	**description**
nbIter	nombre total d'itérations
$\alpha_1, \alpha_2, \alpha_3$	valeurs possibles du coefficient de randomisation
period	période pour la modification du coefficient de randomisation
ttIn et *ttOut*	paramètres fixant la durée de rétention dans les listes taboue

3.6 La liste taboue réactive

En plus des algorithmes SATS et STS décrits ci-dessus, nous avons implanté un autre algorithme que nous appelons *Reactive Tabu Search* (RTS). L'algorithme RTS est une extension de l'algorithme TS dans laquelle la liste taboue est réglée de manière auto-adaptative, selon les principes proposés par (Battiti and Tecchiolli, 1994; Battii and Protasi, 2001) et décrites au sous-chapitre 2.2.8.

La fonction de hachage Pour implanter notre algorithme RTS, nous utilisons une fonction de hachage non injective $H(S)$, et un tableau *NbHits*[ ] de taille *sizeHach* pour stocker les clés de hachage des configurations. Nous notons E l'ensemble de toutes les configurations possibles. La fonction $H(S)$ est définie tel que :

$$\begin{array}{rccl} H: & E & \longrightarrow & [0, sizeHach[\\ & S & \longmapsto & H(S) \end{array}$$

La non-injectivité de notre fonction de hachage $H(S)$ implique qu'une collision correspond à la génération de la même clé, et pas forcément la même configuration : $H(S_1) = H(S_2) \not\Longrightarrow S_1 = S_2$.

La variable *cyclage* Pour détecter les cycles, nous utilisons une variable *cyclage*. Notre contrôle sur la variable *cyclage* se fait toutes les *period* itérations. La valeur de cette variable augmente en fonction du nombre de collisions détectées durant la dernière période. Pour tout entier $p \in [0, sizeHach - 1]$, nous définissons une fonction $T_{iter}(p)$ tel que à une itération *iter* :

$$T_{iter}(p) = |\{i \ \in \ ]iter - period, iter] \ | \ H(S_i) = p\}| \tag{3.7}$$

En utilisant la fonction $T_{iter}(p)$, la variable *cyclage* est définie tel que :

$$cyclage = \sum_{p=0}^{sizeHach-1} T_{iter}(p)^2/period \tag{3.8}$$

Adaptation de la longueur des listes taboues Périodiquement, toutes les *period* itérations, nous établissons un bilan de la valeur de la variable *cyclage*. Si cette valeur est supérieure à un certain *seuil*, nous augmentons la valeur de la variable *tt* en l'augmentant par β_{add}, sinon nous la diminuons par β_{minus} (β_{add} largement supérieure à β_{minus}).

Remarques La génération des clés de hachage ainsi que la détection des collisions se font de façon incrémentale, le coût de ces calculs est négligeable. Pour réaliser un bilan périodique, nous utilisons un tableau *TabIndex*[ ] de taille *period* qui nous permet de ne tenir compte que des configurations générées durant la dernière période.

Paramètres de l'algorithme RTS Les paramètres de l'algorithme RTS sont présentés dans le tableau 3.3.

Tableau 3.3 Paramètres de l'algorithme RTS

paramètre	**description**
tt_0	fixe la valeur initiale de la variable *tt*
β_{add}	un coefficient additif de la longueur de la liste taboue *tt*
β_{minus}	un coefficient soustractif de la longueur de la liste taboue *tt*
period	fixe la période selon laquelle le contrôle sur les cycles se fait
seuil	fixe le seuil de nombre de collisions

Pseudo-code du mécanisme tabou réactif Ci-dessous le pseudo-code de notre mécanisme tabou réactif :

Procédure $RTS()$

$IndexIter := iter \bmod period$

$oldkey := TabIndex[IndexIter]$

$cyclage := cyclage - NbHits[oldkey]^2 \div period$

$NbHits[oldkey] - -$

$cyclage := cyclage + NbHits[oldkey]^2 \div period$

$key := h(S)$

$TabIndex[indexIter] := key$

$cyclage := cyclage - NbHits[key]^2 \div period$

$NbHits[key] + +$

$cyclage := cyclage + NbHits[key]^2 \div period$

if $iter \bmod period = 0$ **then**

  if $cyclage > seuil$ **then**

    $tt := tt + \beta_{add}$

  else

    $tt := tt - \beta_{minus}$

Return $(void)$

CHAPITRE 4 DESCRIPTION DE L'IMPLÉMENTATION

Dans ce chapitre, nous décrivons les structures de données et les algorithmes de bas niveau que nous avons utilisés pour implémenter nos algorithmes SATS et STS.

Dans un algorithme de recherche locale, une itération nécessite d'évaluer plusieurs mouvements et de sélectionner l'un d'entre eux – la manière de sélectionner un mouvement dans les algorithmes SATS et STS est décrite au sous-chapitre 3.3. En fait, la majorité du temps de calcul de l'algorithme est consommée par ces deux tâches. Il est donc critique d'utiliser une implémentation efficace afin de rendre l'exécution de ces deux tâches aussi rapide que possible.

Dans la suite, nous commençons par présenter des notations. Nous décrivons ensuite les structures de données utilisées dans notre implémentation, ainsi que nos algorithmes de bas niveau.

4.1 Notations

Rappels Dans la suite de ce chapitre, nous considérons un exemplaire USCP défini par des entiers n (nombre de sous-ensembles) et m (nombre d'éléments) et une famille $\mathcal{F} = \{M_1...M_n\}$ de sous-ensembles de $M = \{1...m\}$ tels que $\bigcup_{i \in N} M_i = M$, où $N = \{1...n\}$. Les éléments de N sont nommés les variables et ceux de M les contraintes. Pour une contrainte c quelconque, nous nommons V_c l'ensemble des variables qui la couvrent : $V_c = \{M_j \mid c \in M_j, \ \forall M_j \in \mathcal{F}\}$.

Une configuration dans l'espace de recherche est représentée par un sous-ensemble quelconque de N. Le coût $f(S)$ d'une configuration quelconque S est défini comme le nombre de contraintes non couvertes dans S :

$$f(S) = |U(S)| \tag{4.1}$$

où $U(S) = \{i \in M \mid i \notin M_j, \forall j \in S\}$. Nous rappelons qu'il existe deux types de mouvements : des mouvements de suppression et d'insertion. Étant donné un mouvement quelconque m applicable à la configuration S, son score est défini comme son impact sur le score de la configuration :

$$\delta(m) = f(S \oplus m) - f(S) \tag{4.2}$$

où $S \oplus m$ désigne la nouvelle configuration obtenue par l'application de m à S.

Nouvelles notations Dans la suite, nous considérons une configuration de référence S (il s'agit de la configuration courante de l'algorithme). Pour une contrainte $c \in M$, nous notons $Cov(c)$ le nombre de sous-ensembles qui couvrent c : $Cov(c) = |\{c \in M_j \mid j \in S\}|$.

Nous utilisons les structures de données suivantes :

- Pour stocker un exemplaire USCP, nous utilisons deux tableaux de listes d'adjacence $AdjC[\,]$ de taille n et $AdjV[\,]$ de taille m. Pour chaque sous-ensemble $M_j \in \mathcal{F}$, $AdjC[j]$ contient la liste de toutes les contraintes $c \in M_j$ alors que l'ensemble V_c des sous-ensembles qui couvrent la contrainte c est contenu dans $AdjV[c]$.
- Pour stocker la configuration S, nous utilisons une structure de données qui permet d'effectuer une insertion, un retrait et de tester si un sous-ensemble appartient à S ou non en temps constant.

4.2 Calcul efficace du score des mouvements

Une caractéristique importante de notre implémentation est le calcul efficace du score des différents mouvements applicables à la configuration courante. Pour atteindre ce but, nous utilisons des structures de données spécifiques et des algorithmes de bas niveau décrits ci-dessous.

Structures de données utilisées Nous utilisons les structures de données suivantes :

- un tableau nommé $Cov[\,]$ de taille m, qui indique pour chaque contrainte c la valeur de $Cov(c)$;
- un tableau nommé $Delta[\,]$ de taille n ; $Delta[x]$ indique le score $\delta(m)$ du mouvement m consistant à supprimer ou insérer x selon que x appartient ou non à S.

En tout temps, ces structures de données devront refléter la configuration courante. Ces structures de données sont donc initialisées au début de l'algorithme (avant la première itération) en fonction de la configuration initiale. Puis, elles sont mises à jour après chaque mouvement afin de refléter la nouvelle configuration. Les algorithmes d'initialisation et de mise à jour des structures de données sont décrits ci-dessous.

Algorithme d'initialisation La procédure $Init()$ est la procédure appelée au début de l'algorithme pour initialiser nos structures de données en fonction de la solution initiale S_0. Le pseudo-code de la procédure est présenté ci-dessous :

Procédure *Init*()
    for $x = 1..n$ **do**
        $Delta[x] := 0$
    for $c = 1..m$ **do**
        $Cov[c] := 0$
    for $x = 1..n$ **do**
        for $\forall c \in AdjC[x]$ **do**
            $Cov[c] + +$
    for $x = 1..n$ **do**
        if $x \in S_0$ **then**
            for $\forall c \in AdjC[x]$ **do**
                if $Cov[c] = 1$ **then**
                    $Delta[x] + +$
        else
            for $\forall c \in AdjC[x]$
                if $Cov[c] = 0$ **then**
                    $Delta[x] - -$
Return (*void*)

La procédure d'initialisation calcule d'abord les valeurs du tableau *Cov*[ ], puis celles du tableau *Delta*[ ].

Algorithmes de mises à jour Le pseudo-code des deux procédures de mise à jour est présenté ci-dessous :

Procédure *ExecMoveRemoval*(y_m)
    for $\forall c \in AdjC[y_m]$ **do**
        $Cov[c] - -$
        if $Cov[c] = 0$ **then**
            for $\forall x \in AdjV[c]$ **do**
                if $x \neq y_m$ **then**
                    DecremDelta(x)
        else if $Cov[c] = 1$ **then**
            $x :=$ *FindTheSubset*(c, y_m)
            IncremDelta(x)
Return (*void*)

Procédure *ExecMoveInsertion*(y_m)
    for $\forall c \in AdjC[y_m]$
        $Cov[c] + +$
        if $Cov[c] = 1$ **then**
            for $\forall x \in AdjV[c]$ **do**
                if $x \neq y_m$ **then**
                    IncremDelta(x)
        else $Cov[c] = 2$ **then**
            $x :=$ *FindTheSubset*(c, y_m)
            DecremDelta(x)
    Return (*void*)

Dans les deux procédures de mise à jour *ExecMoveInsertion*(y_m) et *ExecMoveRemoval*(y_m), nous pouvons identifier les 3 fonctions suivantes :
– procédure *IncremDelta*(x) : cette procédure incrémente par 1 la valeur $Delta[x]$;
– procédure *DecremDelta*(x) : cette procédure décrémente par 1 la valeur $Delta[x]$;
– procédure *FindTheSubset*(c, y_m) : cette procédure retrouve le sous-ensemble x qui couvre la contrainte c avec le sous-ensemble y_m.
Nous expliquons dans le sous-chapitre 4.3 pourquoi nous utilisons les procédures *IncremDelta*(x) et *DecremDelta*(x) au lieu d'utiliser de simples opérations d'incrémentation et de décrémentation.

4.3 Utilisation de files de priorité

Lors d'une itération, l'algorithme STS calcule le score des différents mouvements et sélectionne l'un d'entre eux (en fonction de la valeur de la température courante α). Ces tâches sont réalisées par les procédures suivantes :

– procédure *Find*() : sélection d'un mouvement selon le critère spécifié ;
– procédure *Update*() : mise à jour du tableau $Delta[\,]$ après l'exécution d'un mouvement.

Les algorithmes présentés dans la section précédente permettent d'implanter très efficacement la procédure de mise à jour. La procédure de sélection d'un mouvement peut alors être implantée aisément en $\mathcal{O}(n)$, simplement en parcourant le tableau $Delta[\,]$: c'est ce que nous appelons la technique de base.

Le temps de calcul relatif de ces deux procédures varie considérablement selon le nombre de variables et, surtout, la densité de l'exemplaire traité. Pour un exemplaire suffisamment

peu dense, la procédure de mise à jour peut devenir beaucoup plus rapide que la procédure de sélection. C'est ce qui se passe pour un nombre important d'exemplaires, notamment les exemplaires *CYC* – voir chapitre 5.

Afin de traiter efficacement ces exemplaires, nous utilisons des structures de données supplémentaires constituées de files de priorité. Notez que ces files de priorité vont effectivement accélérer considérablement la sélection d'un mouvement, mais, en contrepartie, elles vont ralentir la mise à jour. Cependant, le bilan global est largement positif pour la plupart des exemplaires – voir les tests présentés au sous-chapitre 5.5.

Structures de données utilisées Nous utilisons 4 files de priorités :

- *PQIn* : contient les variables entrantes (les éléments de $N - S$) qui sont non taboues ;
- *PQInTS* : contient les variables entrantes (les éléments de $N - S$) qui sont taboues ;
- *PQOut* : contient les variables sortantes (les éléments de S) qui sont non taboues ;
- *PQOutTS* : contient les variables sortantes (les éléments de S) qui sont taboues.

Algorithmes de mises à jour Les deux procédures *IncremDelta*(x) et *DecremDelta*(x), utilisées dans les 2 algorithmes *ExecMoveInsertion*(y_m) et *ExecMoveRemoval*(y_m), ne vont pas seulement incrémenter et décrémenter les valeurs de *Delta*[ ], mais elles vont aussi mettre à jour le score du mouvement dans la file de priorité qui contient le mouvement en question.

4.4 Implémentation des files de priorité

Une file de priorité est classiquement définie comme un type abstrait de données qui permet de mémoriser des éléments, en associant une valeur (clé) à chacun d'entre eux. Une file de priorité permet de réaliser les opérations suivantes : insérer un élément avec une valeur associée ; extraire l'élément ayant la plus grande (ou plus petite) valeur ; tester si la file de priorité est vide ou pas. Il existe au moins deux techniques classiques permettant d'implanter une file de priorité : un tas (*Heap*) et des paquets (*Buckets*). Dans notre implémentation, nous utilisons des paquets, car seule cette technique nous permet d'implanter efficacement la sélection biaisée d'un élément.

Nous implémentons nos files de priorité à l'aide d'une classe nommée *PriorityQueue*. Nous supposons que les éléments qui pourront être insérés dans une file de priorité sont numérotés entre 0 et $n - 1$ et que les valeurs sont comprises entre *minVal* et *maxVal*. Lors de la construction d'une file de priorité, on spécifie :

- la taille des *Buckets* nommée *size_bucket* ;

– l'intervalle des valeurs associées aux éléments $[minVal, maxVal]$;
– le nombre de *Buckets* nommé $nbBck = minVal\text{-}maxVal{+}1$.

Attributs de la classe *PriorityQueue* Pour implanter la classe *PriorityQueue*, nous utilisons les structures de données suivantes (attributs de la classe) :

– deux tableaux $BckNum[\,]$ et $Pos[\,]$ de taille n ;
– un tableau de Buckets (paquets) $Bck[\,]$ de taille $nbBck$.

Chaque Bucket est implanté par un tableau. Pour un élément $x \in PQ$, $BckNum[x]$ indique le Bucket qui contient x et $Pos[x]$ la position de x dans $Bck[BckNum[x]]$.

Méthodes de la classe *PriorityQueue* La classe *PriorityQueue* implémente les méthodes suivantes :

– *Init*() : cette méthode initialise la structure de données à vide ;
– *Insert*(x, val) : cette méthode permet d'insérer un élément x dont la valeur est val ;
– *Remove*(x) : cette méthode permet de supprimer l'élément x ;
– *Update*(x, new_val) : cette méthode met à jour le score d'un élément dans la file en faisant appel aux méthodes *Remove*(x) et *Insert*(x, new_val) ;
– *FindBest*() : cette méthode renvoie un élément de score minimum ;
– *FindMetropolis*(α) : cette méthode renvoie un élément selon la technique biaisée décrite au chapitre 3.3, en utilisant le paramètre de randomisation α.

Nous présentons ci-dessous le pseudo-code des différentes méthodes de la classe *PriorityQueue*.

Procédure *FindBest*()
    indexBucket := Find the best bucket not empty
    index := *rand*(*Bck*[*indexBucket*].*size*())
    m := *Bck*[*indexBucket*][*index*]
    Return(*m*, *Delta*[*m*])

La procédure *FindBest*() commence par identifier le premier paquet non vide puis elle renvoie un élément choisi aléatoirement dans ce paquet.

Procédure **FindMetropolis**(α)
    for $i = 0..nbBck - 1$ **do**
        if $i = 0$ **then**
            $ProbBuck[i] := Bck[i].size()$
        else
            $ProbBuck[i] = ProbBuck[i-1] + (Bck[i].size() \times \alpha^i)$
    $alea = double\_rand(0, ProbBuck[nbBck - 1])$
    $Test := false$
    $i := 0$
    while $i \leq nbBck$ and $Test = false$ **do**
        if $alea \leq ProbBuck[i]$ **then**
            $Test := true$
            $indexBucket := i$
        $i++$
    $index := rand(Bck[indexBucket].size())$
    $m := Bck[indexBucket][index]$
    Return (m, $Delta[m]$)

La procédure *FindMetropolis*(α) commence par affecter un score $ProbBuck[i]$ à chaque paquet i, puis elle sélectionne un paquet selon une probabilité proportionnelle au score calculé. Finalement, elle renvoie un élément choisi aléatoirement dans ce paquet.

4.5 Implémentation de la liste taboue

La liste taboue est implémentée en utilisant un tableau $IsTabuUntil[\ ]$ de taille n. Pour chaque variable x, $IsTabuUntil[x]$ indique jusqu'à quelle itération la variable x est taboue. L'algorithme peut ainsi rendre la variable x taboue par l'opération : $IsTabuUntil[x] := iter + tt$, où $iter$ indique le numéro de l'itération courante et tt (pour tabu tenure) le nombre d'itérations pendant lequel la variable x va rester taboue. On peut ensuite tester si une variable y est taboue : c'est le cas si, et seulement si, $IsTabuUntil[y] < iter$.

CHAPITRE 5 ÉTUDE EXPÉRIMENTALE

Dans ce chapitre, nous présentons les expériences réalisées avec nos algorithmes TS (tabou pur), SATS (hybride) et STS (tabou stochastique). Nous rappelons que ces algorithmes ont été décrits dans le chapitre 3. Ci-dessous, nous commençons par présenter les jeux de données utilisés dans nos tests. Puis, nous décrivons les tests réalisés avec l'algorithme SATS, les résultats obtenus avec les algorithmes TS et STS, les expériences réalisées avec la liste taboue réactive et, finalement, les tests destinés à évaluer notre implémentation.

Pour tous les tests présentés dans ce chapitre, nous avons utilisé une machine Intel I7-2600 CPU @ 3.4Ghz avec 16 Go de mémoire vive. Le langage de programmation utilisé est le C++ et les programmes ont été compilés avec l'option O2.

5.1 Jeux de données utilisés dans nos tests

Pour nos tests, nous avons utilisé les mêmes jeux de données que les principaux travaux de la littérature, notamment (Musliu, 2006; Gao et al., 2015). Les jeux de données utilisés par ces auteurs sont issus de la librairie ORLIB (Beasley, 1990b). Cette librairie contient les familles suivantes :

- Familles *4-6*, *A-E* et *NRE-NRH* : Ces familles contiennent 70 exemplaires qui sont générés aléatoirement. Les familles *4-6* sont issues de (Balas and Ho, 1980), *A-E* de (Beasley, 1987) et *NRE-NRH* de (Beasley, 1990a). Les exemplaires de ces familles sont tous des exemplaires pondérés (SCP) sauf la famille *E*.
- Les exemplaires des familles *CYC* et *CLR* : Ces deux familles contiennent 10 exemplaires proposés dans (Grossman and Wool, 1997). Ils découlent de problèmes combinatoires, à savoir deux questions d'Erdős–Gyárfás (voir Shauger, 1998). Ces exemplaires sont non pondérés.
- Les exemplaires de la famille *Railway* : Cette famille inclut 7 gros exemplaires qui ont été proposés par la compagnie publique italienne des chemins de fer *Ferrovie dello Stato Italiane* et la société italienne de recherche opérationnelle dans le cadre de la compétition *FASTER*. Ces exemplaires sont pondérés.

Dans nos tests, nous avons utilisé tous les jeux présentés ci-dessus, sauf les exemplaires de la famille *Railway* pour lesquels il n'existe pas de résultats obtenus avec des métaheuristiques – seulement des résultats obtenus avec des heuristiques lagrangiennes (voir Beasley, 1990a; Caprara et al., 1999). Certains de ces jeux de données sont pondérés : pour les traiter avec

nos algorithmes, nous les avons transformés en jeux non pondérés (USCP) en fixant toutes les pondérations à 1. Les caractéristiques des jeux de données sont résumées dans les tableaux 5.1 et 5.2. Certains jeux de données sont regroupés en familles, tous les jeux d'une même famille ayant les mêmes caractéristiques. Le tableau 5.1 décrit les jeux de données regroupés en familles. Dans ce tableau, les noms des colonnes sont explicites. Par exemple, la première ligne correspond à la famille nommée "*4*". Cette famille contient 5 jeux pondérés. Pour chaque jeu de cette famille, on a $m = 200$ (nombre d'éléments) $n = 1000$ (nombre de sous-ensembles) et la densité vaut 2 %. Cette famille provient de la librairie nommée ORLIB. Ces jeux sont aléatoires et l'optimum de chacun des jeux est connu. Dans ces 2 tableaux, la colonne U/W permet d'identifier la version du problème de l'exemplaire : U pour non pondérés (USCP) et W pour pondérés (SCP).

Le tableau 5.2 décrit les jeux de données qui ne sont pas regroupés en familles (jeux individuels).

5.2 Expériences réalisées avec l'algorithme SATS

Dans ce sous-chapitre, nous présentons des expériences réalisées avec l'algorithme SATS. L'objectif de ces tests consiste à comprendre l'influence des paramètres (fixant la température et la longueur des listes taboues) sur le comportement de l'algorithme et la qualité des solutions. En particulier, nous souhaitons déterminer comment régler les paramètres de l'algorithme.

Il faut noter que, dans les expériences présentées dans ce sous-chapitre, nous utilisons l'algorithme SATS dans sa version décision, car cela tend à simplifier l'analyse. Dans cette version, on fixe la taille de la solution (paramètre *kTarget*) et l'algorithme tente de minimiser la pénalité. L'algorithme s'arrête dès que la pénalité vaut 0 (cas qualifié de succès) ou lorsqu'il a épuisé le nombre d'itérations (cas qualifié d'échec). L'algorithme renvoie la meilleure configuration trouvée (celle dont la pénalité est la plus faible). Nous allons réaliser des séries d'exécutions sur chaque exemplaire testé et nous allons considérer deux mesures de performances : la moyenne du coût (pénalité) des configurations trouvées et le pourcentage de succès. L'algorithme SATS dans sa version décision utilise les paramètres suivants : *kTarget*, *nbIter*, *tt*, α et β. Le paramètre *tt* sert à régler les longueurs de deux listes taboues en appliquant les équations $ttIn = tt$ et $ttOut = 2 \times tt$. Le paramètre β sert à contrôler la décroissance de la température.

Les tests présentés dans cette section ont été réalisés avec *CYC08* avec une taille de solution égale à la valeur du meilleur résultat dans la littérature $kTarget = 342$. Pour chaque exécution,

Tableau 5.1 Jeux de données (groupés en familles)

Famille	U/W	m	n	d (%)	Origine	Nombre de jeux	Type	Optimum
4	w	200	1000	2	ORLIB	10	Aléatoire	Connu
5	w	200	2000	2	ORLIB	10	Aléatoire	Connu
6	w	200	1000	5	ORLIB	5	Aléatoire	Connu
A	w	300	3000	2	ORLIB	5	Aléatoire	Connu
B	w	300	3000	5	ORLIB	5	Aléatoire	Connu
C	w	400	4000	2	ORLIB	5	Aléatoire	Connu
D	w	400	4000	5	ORLIB	5	Aléatoire	Connu
E	u	50	500	20	ORLIB	5	Aléatoire	Connu
NRE	w	500	5000	10	ORLIB	5	Aléatoire	Inconnu
NRF	w	500	5000	20	ORLIB	5	Aléatoire	Inconnu
NRG	w	1000	10000	2	ORLIB	5	Aléatoire	Inconnu
NRH	w	1000	10000	5	ORLIB	5	Aléatoire	Inconnu

Tableau 5.2 Jeux de données (individuels)

Exemplaire	U/W	m	n	d (%)	Origine	Type	Optimum
CLR10	u	511	210	12.3	ORLIB	Combinatoire	Inconnu
CLR11	u	1023	330	12.4	ORLIB	Combinatoire	Inconnu
CLR12	u	2047	495	12.5	ORLIB	Combinatoire	Inconnu
CLR13	u	4095	715	12.5	ORLIB	Combinatoire	Inconnu
CYC6	u	240	192	2.1	ORLIB	Combinatoire	Connu
CYC7	u	672	448	0.9	ORLIB	Combinatoire	Inconnu
CYC8	u	1792	1024	0.4	ORLIB	Combinatoire	Inconnu
CYC9	u	4608	2304	0.2	ORLIB	Combinatoire	Inconnu
CYC10	u	11520	5120	0.08	ORLIB	Combinatoire	Inconnu
CYC11	u	28160	11264	0.04	ORLIB	Combinatoire	Inconnu

le nombre maximum d'itérations (*nbIter*) vaut 2×10^6 itérations. Lors d'une série d'exécutions, le nombre total d'itérations est fixé à 50 fois la valeur du paramètre *nbIter* (donc 100 millions). Ainsi, si l'algorithme échoue à chaque exécution, il réalise 50 exécutions, mais il peut réaliser beaucoup plus d'exécutions en cas de succès. En réalisant ainsi un grand nombre d'exécutions, nous obtenons des moyennes précises et des courbes lisses.

Dans la suite, nous commençons par étudier l'algorithme de recuit simulé. Puis, nous considérons le cas de SATS à température constante. Finalement, nous terminons avec l'algorithme SATS avec décroissance de la température.

5.2.1 Comportement de l'algorithme de recuit simulé

Nous commençons par considérer le cas de l'algorithme de recuit simulé à température constante. Puis nous analysons le cas de l'algorithme de recuit simulé avec décroissance de la température.

Algorithme de recuit simulé à température constante Nous présentons les résultats de tests réalisés avec l'algorithme de recuit simulé à température constante. Le but de l'expérience réalisée consiste à analyser le comportement de cet algorithme en fonction de la température – ou plutôt de la valeur du paramètre α jouant le même rôle que la température. Dans cette expérience, nous utilisons donc l'algorithme SATS sans liste taboue ($tt = 0$) et sans décroissance de température ($\beta = 0$).

La figure 5.1 affiche des profils d'exécution obtenus à différentes températures : (α = 0, 0.02, 0.14, 0.25, 0.6). Chaque courbe correspond à une exécution unique réalisée avec une température particulière. La courbe représente un profil de l'exécution considérée, c'est-à-dire qu'elle indique l'évolution du coût de la configuration courante en fonction du nombre d'itérations.

La courbe en vert correspond à la valeur la plus élevée du paramètre de température (α = 0.6). Nous observons que cette courbe présente de grandes oscillations et que la fonction de coût reste à une valeur élevée. La courbe en bleu correspond à une valeur plus petite de α ($\alpha = 0.25$). Nous observons que cette courbe présente des oscillations plus faibles et atteint des valeurs plus basses. Ce phénomène est encore plus marqué avec la courbe en jaune ($\alpha = 0.14$) ; avec cette valeur du paramètre, une solution de coût nulle est atteinte au bout d'environ 900,000 itérations. La courbe en gris correspond à une valeur encore plus faible du paramètre ($\alpha = 0.02$). Nous observons que, non seulement la courbe oscille très peu, mais qu'elle tend à se bloquer – à rester stationnaire durant un nombre élevé d'itérations consécutives. Cette courbe reste bloquée à une valeur d'environ 15 au bout d'un million

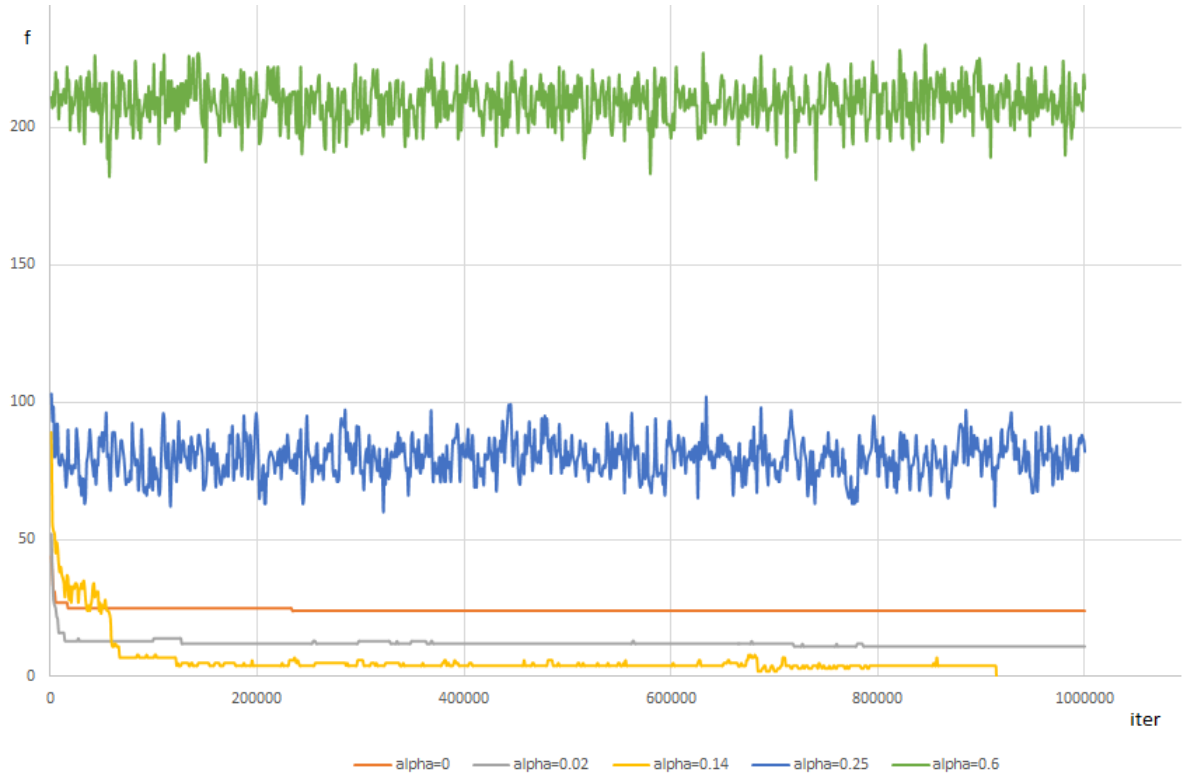

Figure 5.1 Algorithme de recuit simulé à température constante : profils d'exécution obtenus à différentes températures

d'itérations. La courbe en orange correspond à une valeur nulle du paramètre α – l'algorithme de recuit simulé s'apparente donc dans ce cas à un algorithme de descente. Bien que cette courbe commence par décroître très fortement durant les premières itérations, elle se bloque rapidement à une valeur élevée égale environ à 25.

Nous pouvons interpréter ces observations de la manière suivante. Avec des valeurs trop élevées de la température, l'algorithme accepte trop facilement des mouvements qui dégradent le coût de la configuration, ce qui l'empêche d'atteindre de bonnes solutions. Inversement, avec des valeurs trop basses, l'algorithme éprouve de la difficulté à échapper aux bassins d'attraction des optima locaux.

La figure 5.2 affiche des résultats obtenus en réalisant avec l'algorithme de recuit simulé (à température constante) des séries d'exécutions à différentes températures. La figure affiche le coût moyen de la configuration renvoyée par l'algorithme en fonction de la température.

Nous observons que la courbe présente un profil en U : le coût des solutions décroît jusqu'à une valeur du paramètre de température égale à 0.08, puis remonte ensuite. On peut supposer qu'il existe une valeur idéale du paramètre de température α égale à environ 0.08. Lorsque la valeur du paramètre s'éloigne de la valeur idéale, que ce soit par valeurs supérieures ou inférieures, la performance se dégrade.

Analyse du recuit simulé avec température décroissante Dans le recuit simulé standard, la température décroît. Le schéma de refroidissement le plus courant est le schéma à décroissance géométrique, dans lequel on réduit la température à chaque itération en la multipliant par un coefficient $1 - \beta$, où le taux de décroissance β ($0 < \beta < 1$) est une valeur proche de zéro. Dans la suite, nous analysons le comportement et la performance que nous pouvons attendre d'un tel algorithme en fonction de trois paramètres : les températures initiale et finale, ainsi que le coefficient de décroissance.

Influence des paramètres Pour comprendre l'impact de ces trois paramètres, nous pouvons nous référer aux expériences rapportées précédemment aux figures 5.1 et 5.2. Nous avons vu que le recuit simulé est efficace et produit de bonnes solutions dans un certain intervalle de températures, intervalle que nous appelons l'intervalle efficace et que nous notons $[\alpha_{inf}\ ,\ \alpha_{sup}]$, même si nous savons que les bornes de cet intervalle ne sont pas définies de manière tranchée. Si nous supposons que la température décroît en partant d'une valeur suffisamment élevée (supérieure à α_{sup}) et que nous l'arrêtons à une température suffisamment basse (inférieure à α_{inf}), l'algorithme va passer par trois phases : durant la phase initiale, la température sera trop élevée (supérieure à α_{sup}), puis appropriée (comprise entre α_{inf} et α_{sup}), et finalement

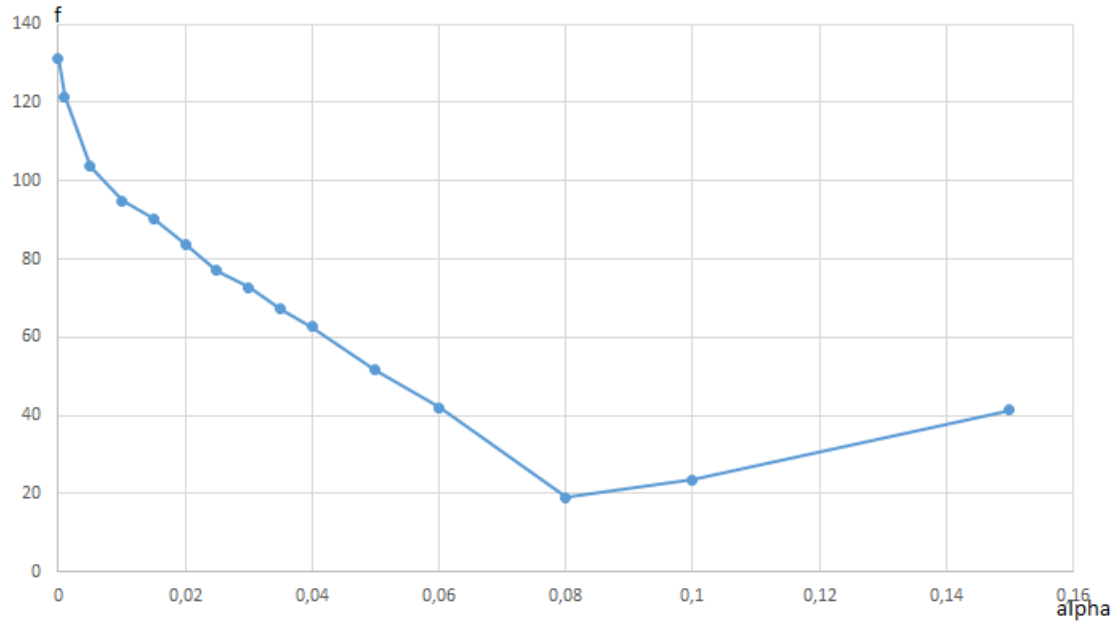

Figure 5.2 Recuit simulé à température constante : coût moyen des solutions en fonction de la température

trop basse (inférieure à α_{inf}). Pour régler les paramètres de l'algorithme, il suffit de choisir une température initiale située au-dessus de la valeur seuil α_{sup}, mais pas trop, pour éviter de perdre trop de temps au début à des températures trop élevées. Idem pour la température finale qu'il convient de régler un peu au-dessous de la valeur seuil α_{inf}. Dans ces conditions, nous sommes assurés que, à un moment ou à un autre, l'algorithme va utiliser les valeurs contenues dans l'intervalle efficace.

Robustesse de l'algorithme D'après ce raisonnement, le choix des valeurs des paramètres de température finale et initiale n'est pas très critique, le risque étant de perdre du temps inutilement à haute ou basse température. Ajoutons que, plutôt que de fixer la température finale, il est plus simple d'arrêter l'algorithme au bout d'un certain nombre d'itérations sans amélioration du meilleur score atteint par la fonction de coût. Le rôle du paramètre de décroissance β est assez simple à interpréter : plus la décroissance est faible (lente), plus l'algorithme va rester longtemps dans l'intervalle efficace de température : ceci aura pour effet d'augmenter le nombre d'itérations (et donc le temps de calcul), mais cela tendra à améliorer l'efficacité de l'algorithme.

Le raisonnement ci-dessus équivaut au modèle suivant : l'efficacité de l'algorithme dépend du nombre d'itérations durant lesquelles la température se trouve dans l'intervalle efficace et s'améliore lorsque celui-ci augmente. On peut observer que, d'après ce modèle, on s'attend

à ce que la performance varie peu si la température initiale change (en restant suffisamment élevée) alors que le taux de décroissance et la température finale restent les mêmes.

5.2.2 Comportement de l'algorithme tabou

La figure 5.3 a été obtenue avec l'algorithme tabou pur obtenu dans SATS en réglant la valeur du paramètre de température à 0. Dans cette expérience, on a fait varier le paramètre tabou (tt = 0, 7, 14, 21, 28, 35, 45, 55, 65, 75, 85).

La courbe affiche le coût moyen des solutions renvoyées par l'algorithme en fonction de la valeur du paramètre tabou *tt*. On observe à nouveau un profil en U avec un minimum d'environ 5 atteint lorsque $tt = 45$. Si nous comparons cette courbe à la courbe précédente 5.2, nous remarquons que l'algorithme tabou permet d'atteindre des valeurs de la fonction de coût qui sont plus basses que celles obtenues avec le recuit simulé avec, en particulier, un minimum de 5 pour tabou contre 20 pour le recuit simulé. La supériorité de tabou sur le recuit simulé que nous constatons dans nos expériences réalisées pour le problème USCP n'est pas tellement étonnante ; en effet, différents auteurs ont rapporté dans la littérature que, pour un même problème et en utilisant le même voisinage, tabou est souvent plus efficace que le recuit simulé. C'est le cas pour le problème de coloriage de graphe – (voir Chams et al., 1987; Hertz and de Werra, 1987).

5.2.3 Comportement de l'algorithme SATS à température constante

Le but de l'expérience présentée consiste à analyser le comportement de l'algorithme SATS à température constante. Elle consiste à observer la performance réalisée par l'algorithme lorsqu'on fait varier le paramètre de température (α) et le paramètre tabou (tt). Dans cette expérience, nous utilisons l'algorithme SATS sans décroissance de température ($\beta = 0$) et nous testons l'algorithme SATS avec les valeurs suivantes des paramètres : (α = 0, 0.001, 0.002, 0.005, 0.01, 0.015, 0.02, 0.025, 0.03, 0.035, 0.04, 0.05, 0.06, 0.08, 0.1, 0.15) et (tt = 0, 7, 14, 21, 28, 35, 45, 55).

La figure 5.4 affiche le coût moyen des solutions en fonction de la température, pour différentes valeurs du paramètre tabou. Sur la figure, chaque courbe correspond à une valeur du paramètre *tt*. On note que la courbe en bleu clair ($tt = 0$) correspond donc au recuit simulé ; cette courbe est bien la même que la courbe unique affichée sur la figure 5.2.

Décalage de la température optimale Sur la figure 5.4, nous observons que chacune des courbes présente un profil en U. Pour une valeur particulière de *tt*, notons $f^*(tt)$ la valeur

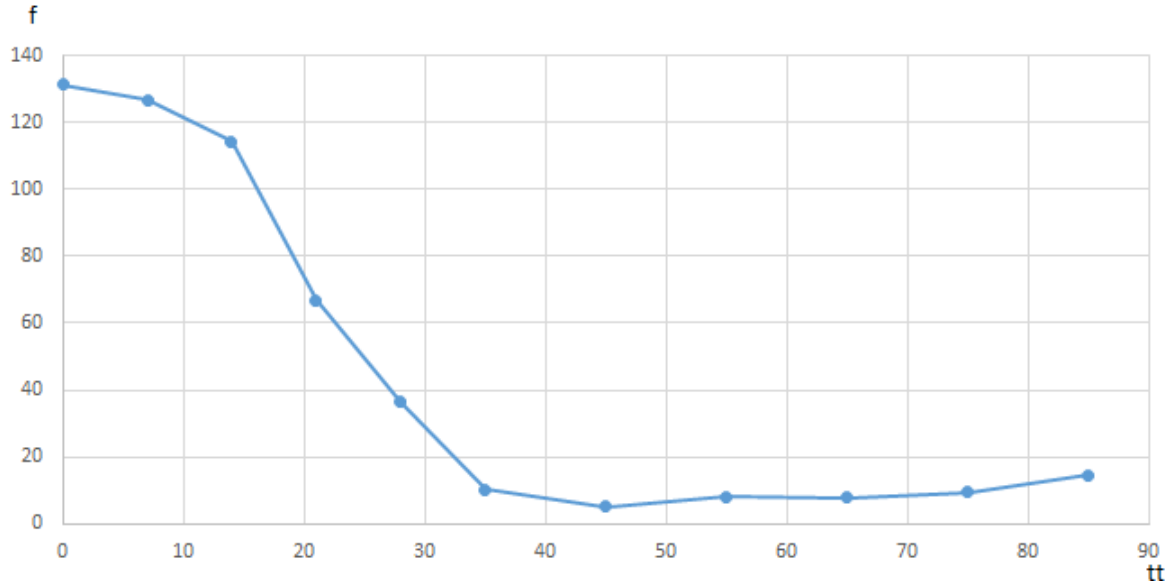

Figure 5.3 Algorithme tabou : coût moyen des solutions en fonction du paramètre tabou

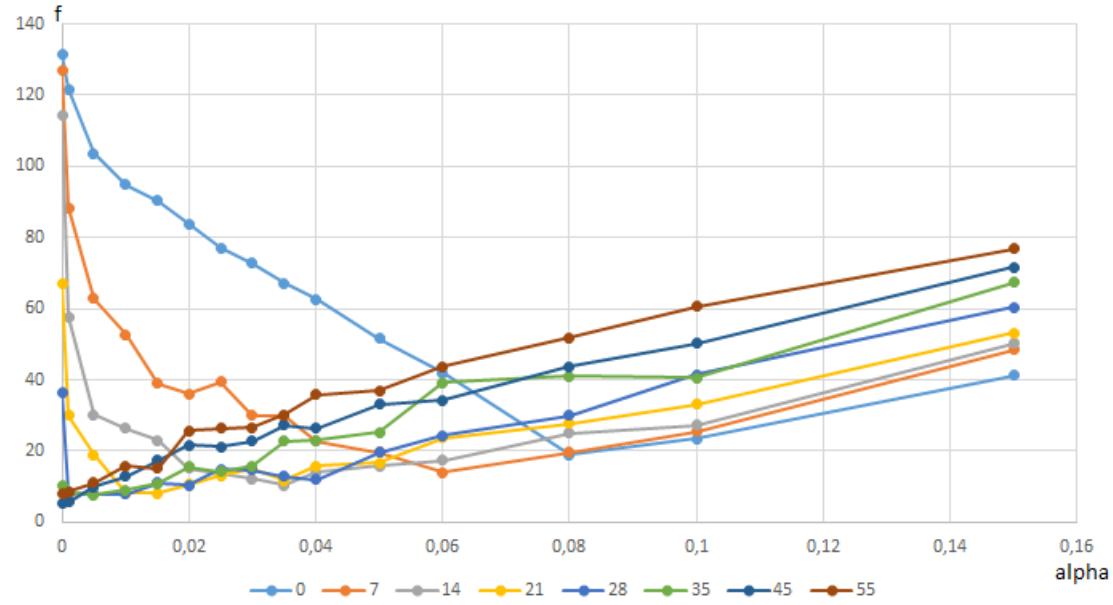

Figure 5.4 Algorithme SATS à température constante : coût moyen des solutions en fonction de la température, pour différentes valeurs du paramètre tabou

minimum atteinte par le coût, et par $\alpha^*(tt)$ la valeur de la température qui permet d'atteindre cette valeur. $\alpha^*(tt)$ représente donc la valeur optimale de la température qui correspond à une valeur tt donnée. Par exemple, en observant la courbe de couleur bleu clair ($tt = 0$), nous pouvons remarquer que $\alpha^*(0)$ vaut environ 0.08 alors que $f^*(0) = 20$. Si nous observons les différentes courbes pour des valeurs croissantes de tt en partant de $tt = 0$, nous remarquons que l'abscisse de l'extremum se déplace vers la gauche. Cela signifie que, lorsque tt augmente, la valeur optimale $\alpha^*(tt)$ du paramètre de température décroît.

Ce phénomène semble pouvoir s'expliquer aisément à l'aide du modèle simplifié suivant. La liste taboue et la randomisation de la solution ont toutes les deux un effet de diversification. La diversification produite augmente lorsque α ou tt augmente. On peut supposer qu'il existe un niveau idéal de diversification qui permet une efficacité maximum de l'algorithme. Il est donc normal que, si on augmente l'un des deux paramètres, il faille diminuer l'autre pour atteindre le niveau optimal de diversification.

Inefficacité de l'algorithme du recuit simulé pur Si nous observons les différentes courbes pour des valeurs croissantes de tt en partant de $tt = 0$, nous remarquons que l'ordonnée de l'extremum ($f^*(tt)$) est inférieure ou égale à 10 pour $tt \geq 14$, mais supérieure pour les deux plus petites valeurs $tt = 0$ et $tt = 7$. Ceci signifie que l'algorithme SATS peut obtenir de bons résultats pour les différentes valeurs de tt (pourvu que le réglage de α soit correct), sauf lorsque la valeur du paramètre tabou est trop faible – le pire cas correspondant au recuit simulé pur ($tt = 0$).

La figure 5.5 affiche le taux de succès en fonction de la température, pour différentes valeurs du paramètre tabou.

On observe que les courbes sont en U inversé – il est normal que le U soit inversé puisqu'une efficacité élevée de l'algorithme correspond à une valeur basse du coût des solutions mais, au contraire, à une valeur élevée du taux de succès. Ces résultats confirment ceux observés sur la figure précédente 5.4. En particulier, on observe à nouveau une diminution de la valeur optimale de α lorsque tt augmente ainsi qu'une moindre efficacité de l'algorithme pour les plus petites valeurs de tt – avec un taux d'efficacité maximum de 20 % et 45 % lorsque $tt = 0$ et $tt = 7$ respectivement, et un taux d'efficacité maximum supérieur à 65 % pour des valeurs plus grandes.

Les figures 5.6 et 5.7 affichent le coût moyen des solutions et le taux de succès en fonction du paramètre tabou, pour différentes valeurs de la température. Elles fournissent donc le même type d'information que les figures 5.4 et 5.5, mais en inversant le rôle joué par les paramètres α et tt. On peut observer à nouveau sur ces figures les phénomènes déjà observés sur les

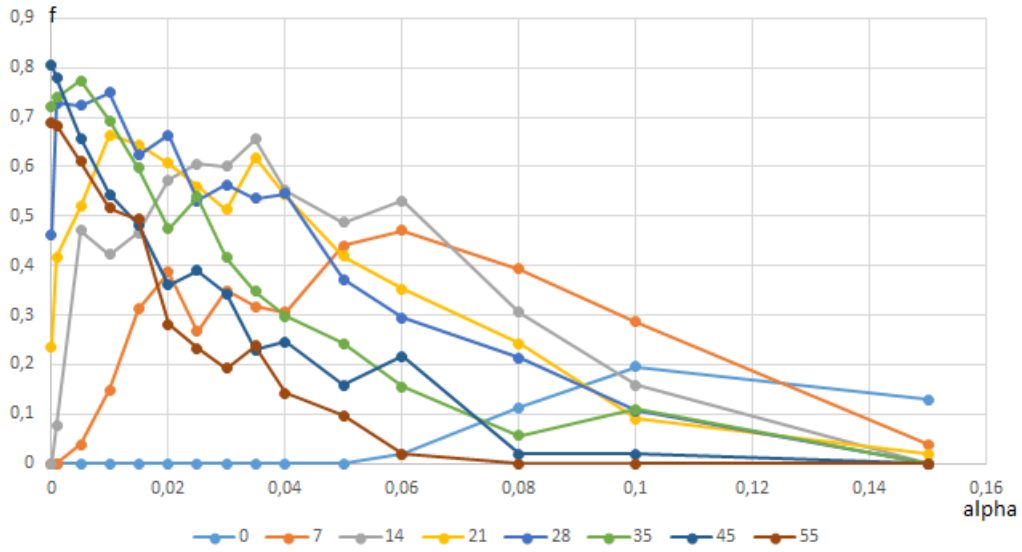

Figure 5.5 Algorithme SATS à température constante : taux de succès en fonction de la température, pour différentes valeurs du paramètre tabou

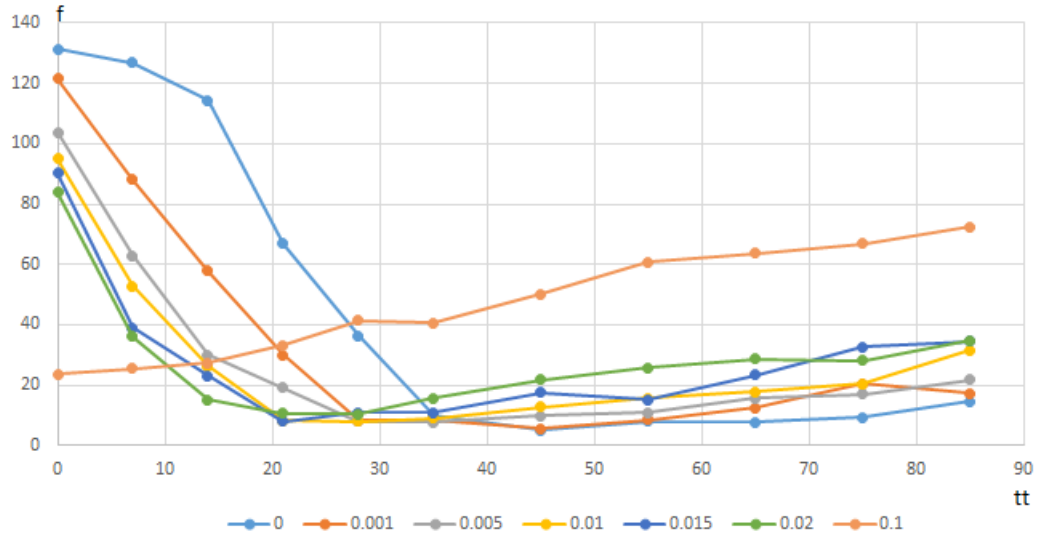

Figure 5.6 Algorithme SATS à température constante : coût moyen des solutions en fonction du paramètre tabou, pour différentes valeurs de la température

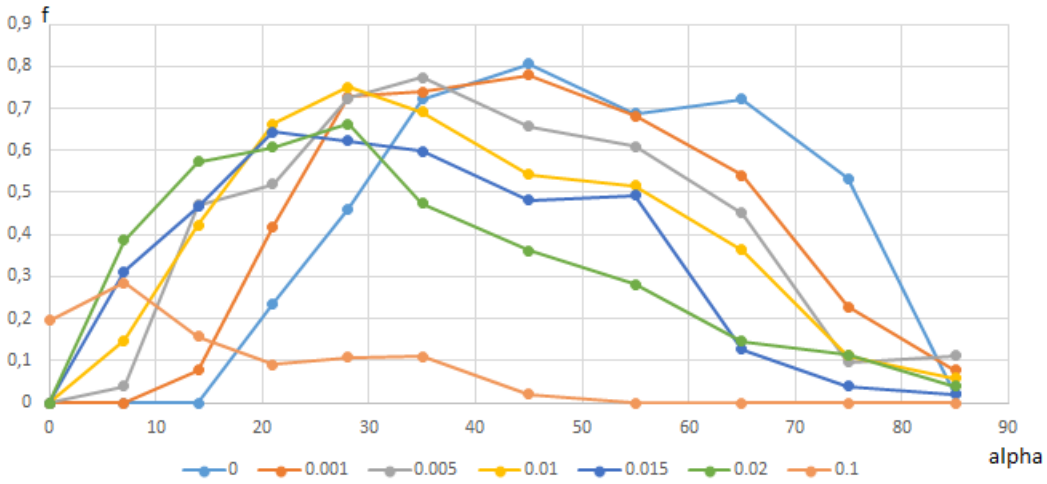

Figure 5.7 Algorithme SATS à température constante : taux de succès en fonction du paramètre tabou, pour différentes valeurs de la température

figures 5.4 et 5.5.

Le tableau 5.8 affiche le coût moyen des solutions en fonction du paramètre tabou et de la température. Les valeurs marquées en vert correspondent à une valeur du coût inférieure à 10. Nous observons que, pour chacune des colonnes qui correspondent aux valeurs de *tt* comprises entre 21 et 45, deux ou trois lignes apparaissent en vert. Ceci signifie que, pour ces différentes valeurs de *tt*, il existe une plage assez large de valeurs de α qui permettent d'obtenir de bonnes solutions. Ceci indique que ces différentes valeurs de *tt* permettent un réglage robuste du paramètre α.

Le tableau 5.9 affiche le taux de succès en fonction du paramètre tabou et de la température.

	0	7	14	21	28	35	45	55	65	75	85
0	131,44	126,74	114,36	66,8364	36,2923	10,1346	5,2	8,096	7,72222	9,4	14,5098
0,001	121,58	88,22	57,75	30,0149	8,22826	8,4958	5,61429	8,55556	12,4588	20,4035	17,2308
0,005	103,62	62,9608	30,1324	19,04	7,875	7,69343	9,87719	11,019	15,6901	17	21,8333
0,01	94,92	52,7593	26,3099	8,13483	7,92857	9,17949	12,8043	15,7363	17,7879	20,3333	31,5769
0,015	90,28	39,0328	23,0274	8,06667	11,0323	10,8824	17,4699	15,2192	23,3636	32,6471	34,4118
0,02	83,88	36,129	15,0533	10,5714	10,3895	15,575	21,6528	25,8281	28,5636	28,1509	34,902
0,025	77,08	39,3833	13,8519	13,1667	14,9277	14,1648	21,1299	26,35	29,7679	35,62	42,2
0,03	72,76	30	12,2556	15	14,7011	15,7922	22,9143	26,7368	33,2593	39,9216	45,72
0,035	67,22	29,873	10,4368	11,7732	12,8452	22,75	27,2787	30,2034	35,3019	38,6863	49,5
0,04	62,64	22,7581	13,9211	15,8193	12	23	26,4426	35,9464	41,1731	41,902	46,7
0,05	51,58	19,6667	15,9054	16,6351	19,7	25,3871	33,1053	37,0192	44,38	47,16	53,02
0,06	42,0196	14,1029	17,3418	23,7059	24,4706	39,2241	34,25	43,8039	48,08	51,44	56,14
0,08	19	19,6364	25,1129	27,8548	29,9836	41	43,8431	51,94	57,98	60,86	63,2
0,1	23,5714	25,5397	27,386	33,2727	41,5179	40,5818	50,2745	60,66	63,68	66,88	72,46
0,15	41,3333	48,5	50,36	53,1569	60,52	67,38	71,56	76,94	85,32	95,64	106,5

Figure 5.8 Algorithme SATS à température constante : coût moyen des solutions en fonction du paramètre tabou et de la température

	0	7	14	21	28	35	45	55	65	75	85
0	0	0	0	0,23636364	0,46153846	0,72115385	0,80666667	0,688	0,72222222	0,53333333	0,01960784
0,001	0	0	0,07692308	0,41791045	0,72826087	0,7394958	0,77857143	0,68253968	0,54117647	0,22807018	0,07692308
0,005	0	0,03921569	0,47058824	0,52	0,72321429	0,77372263	0,65789474	0,60952381	0,45070423	0,09615385	0,11111111
0,01	0	0,14814815	0,42253521	0,66292135	0,75	0,69230769	0,54347826	0,51648352	0,36363636	0,11111111	0,05769231
0,015	0	0,31147541	0,46575342	0,64444444	0,62365591	0,59803922	0,48192771	0,49315068	0,12727273	0,03921569	0,01960784
0,02	0	0,38709677	0,57333333	0,60714286	0,66315789	0,475	0,36111111	0,28125	0,14545455	0,11320755	0,03921569
0,025	0	0,26666667	0,60493827	0,55952381	0,53012048	0,53846154	0,38961039	0,23333333	0,19642857	0	0
0,03	0	0,34920635	0,6	0,5125	0,56321839	0,41558442	0,34285714	0,19298246	0,11111111	0,01960784	0
0,035	0	0,31746032	0,65517241	0,6185567	0,53571429	0,34722222	0,2295082	0,23728814	0,0754717	0,01960784	0
0,04	0	0,30645161	0,55263158	0,54216867	0,54545455	0,29850746	0,24590164	0,14285714	0,03846154	0,01960784	0
0,05	0	0,43939394	0,48648649	0,41891892	0,37142857	0,24193548	0,15789474	0,09615385	0	0	0
0,06	0,01960784	0,47058824	0,53164557	0,35294118	0,29411765	0,15517241	0,21666667	0,01960784	0	0	0
0,08	0,11320755	0,39393939	0,30645161	0,24193548	0,21311475	0,05660377	0,01960784	0	0	0	0
0,1	0,19642857	0,28571429	0,15789474	0,09090909	0,10714286	0,10909091	0,01960784	0	0	0	0
0,15	0,12962963	0,03846154	0	0,01960784	0	0	0	0	0	0	0

Figure 5.9 Algorithme SATS à température constante : taux de succès en fonction du paramètre tabou et de la température

Les valeurs marquées en vert correspondent à un taux de succès supérieur ou égal à 60 %. Comme dans le tableau précédent 5.8, on observe pour les valeurs de tt comprises entre 21 et 45 qu'il existe une plage assez large de valeurs de α qui permettent une bonne efficacité de l'algorithme.

5.2.4 Comportement de l'algorithme SATS avec température décroissante

Le but de l'expérience présentée dans cette section consiste à analyser l'évolution de la performance de l'algorithme SATS lorsqu'on fait varier la température initiale, mais que la température finale et le taux de décroissance sont fixés. Dans cette expérience, la température finale α_{finale} est fixée à 0.0001 et le taux de décroissance β à 0.98 .

La figure 5.10 affiche le taux de succès de l'algorithme en fonction de la température initiale, pour différentes valeurs du paramètre tabou.

Perte d'efficacité pour les hautes températures initiales Sur la figure, nous observons pour chacune des courbes que le taux de succès présente des oscillations pour les températures initiales basses, puis que le taux de succès décroît ensuite à partir d'un certain seuil. Ce seuil dépend de la valeur du paramètre tt mais il est dans les trois cas affichés inférieur à 0.05.

Par exemple, pour $tt = 21$, le taux de succès s'élève à plus de 50 % pour $\alpha = 0.04$ et à seulement 20 % pour $\alpha = 0.1$. Nous observons que, dans le deuxième cas, l'algorithme va commencer par réaliser une série d'itérations pour atteindre la température $\alpha = 0.04$, puis qu'il va réaliser exactement le même travail que dans le premier cas pour les valeurs de température comprises entre $\alpha = 0.04$ et la température finale $\alpha_{finale} = 0.0001$. Le

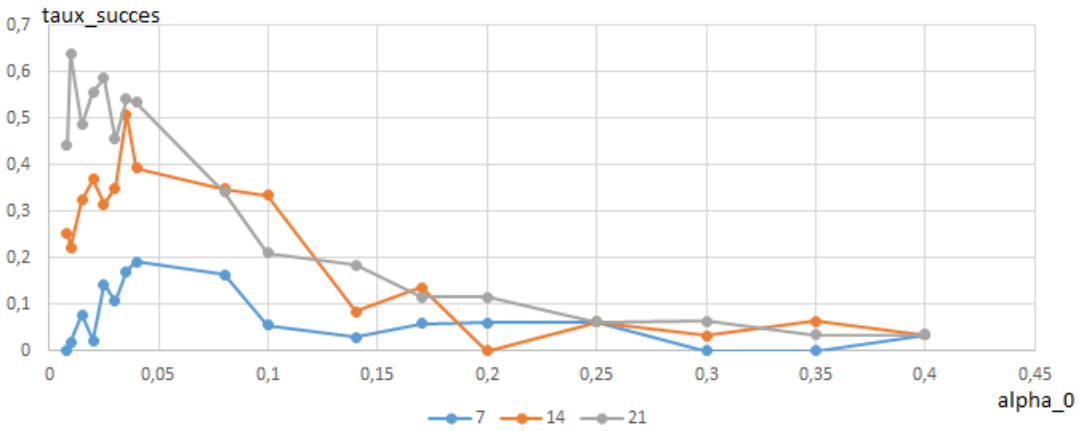

Figure 5.10 Algorithme SATS à température décroissante : taux de succès en fonction de la température initiale, pour différentes valeurs du paramètre tabou

phénomène observé est paradoxal, car, en commençant à une température plus haute, tout se passe comme si l'algorithme faisait simplement plus de travail au tout début de la recherche.

Tentative d'explication Le phénomène observé ci-dessus de perte d'efficacité de notre algorithme SATS pour les hautes températures initiales semble paradoxal. Une explication possible, que nous n'avons pas cherché à valider, est la suivante : il pourrait exister dans le paysage de recherche des régions d'altitude élevée dans lesquelles se trouvent des dépressions ou des vallées profondes d'où la configuration ne peut pas parvenir à s'extraire. En s'exécutant à haute température, l'algorithme passe davantage de temps dans les régions de haute altitude et augmente ainsi le risque de se trouver piégé dans une telle dépression.

Réglage approprié des paramètres de l'algorithme SATS Notre idée initiale était d'utiliser un algorithme SATS à température décroissante afin de rendre robuste le réglage de l'algorithme. En fixant la température initiale à une valeur élevée, il semblait que le seul risque était celui d'allonger le temps d'exécution. Malheureusement, nous venons d'observer que, pour que l'algorithme SATS soit efficace, la température doit rester très basse. Sur la base de ces observations, nous avons développé une variante de l'algorithme SATS sans décroissance de la température. Cette variante est appelée l'Algorithme Tabou Stochastique (STS). Cet algorithme a été décrit dans la section 3.5. Des expériences réalisées avec cet algorithme seront présentées dans le prochain sous-chapitre du mémoire.

5.3 Résultats obtenus avec les algorithmes TS et STS

Dans ce sous-chapitre, nous présentons les résultats obtenus avec notre algorithme STS. Nous ajoutons aussi des résultats obtenus avec TS et des comparaisons avec l'algorithme RWLS proposé dans (Gao et al., 2015). Nous rappelons que les algorithmes renvoient le recouvrement trouvé de cardinalité minimum. Nous appelons résultat ou performance la taille de ce recouvrement.

5.3.1 Réglage des paramètres des algorithmes TS et STS

Comme expliqué au chapitre 3, nos algorithmes TS et STS utilisent deux listes taboues, une pour les variables entrantes et une pour les variables sortantes. Les longueurs *ttIn* et *ttOut* de ces deux listes sont deux paramètres de l'algorithme que nous devons donc fixer. Nous cherchons une formule générale qui pourrait dépendre des caractéristiques du problème (n, m et la densité d) et de la taille de la solution. Pour trouver cette formule, nous avons procédé en deux étapes :

- Durant l'étape 1, nous avons sélectionné un échantillon de jeux de données présentant des caractéristiques variées en matière de taille, densité ou type. Pour chacun de ces exemplaires, nous trouvons empiriquement des combinaisons de valeurs acceptables pour les paramètres *ttIn* et *ttOut*.
- Durant l'étape 2, nous cherchons à découvrir une formule qui donne des valeurs s'approchant le mieux possible des valeurs trouvées pour chaque exemplaire durant l'étape 1.

Nous avons obtenu les formules suivantes qui ne dépendent que de la taille de la solution :

$$ttIn = 10 + \frac{|S|}{30} \tag{5.1}$$

$$ttOut = \frac{|S|^{3/4}}{3} + 1 \tag{5.2}$$

Pour l'algorithme STS, nous avons procédé de manière analogue. Nous sommes parvenus à trouver des équations linéaires qui ne dépendent que la taille de la solution.

$$ttIn = 5 + 0.0015 \times |S| \tag{5.3}$$

$$ttOut = 1.5 \times ttIn \tag{5.4}$$

5.3.2 Tests systématiques réalisés avec les algorithmes TS et STS

Nous avons réalisé des tests systématiques avec nos algorithmes TS et STS sur l'ensemble des jeux de données. Pour fixer les paramètres des algorithmes, nous avons utilisé les formules 5.1, 5.2, 5.3 et 5.4 trouvées au sous-chapitre 5.3.1. L'implémentation des algorithmes est l'implémentation standard A+ (avec files de priorité) pour tous les jeux, sauf pour les jeux denses tels que $d > 0.2$. Pour ces tests, le critère d'arrêt est le nombre d'itérations. Celui-ci a été fixé à 200 millions pour les jeux combinatoires *CYC* et *CLR* et à 60 millions pour les jeux aléatoires*(4-6, A-E, NRE, NRF, NRG et NRH)*. Ces limites sont les mêmes que celles fixées dans (Gao et al., 2015). Nous avons fixé le nombre d'exécutions à 20, mais avec une limite de 10 heures au total par jeu de données.

Le tableau de résultats A.1 est présenté dans l'annexe A. Ce tableau indique, pour chacun des jeux de données, les performances de TS et de STS, à savoir le meilleur résultat obtenu (*Min*), la moyenne des résultats de toutes les exécutions (*Avg*), le nombre de fois que l'algorithme atteint le meilleur résultat (*Succ*), le temps CPU nécessaire pour effectuer le nombre total d'itérations (*CpuT*) et la moyenne de nombre d'itérations nécessaire pour atteindre le meilleur résultat (*IBest*). Avec ce tableau, notre but consiste à fournir des informations détaillées permettant des comparaisons ultérieures. Nous avons ajouté la valeur minimum obtenue sur 10 exécutions par l'algorithme RWLS en utilisant le même critère d'arrêt. Les valeurs affichées dans cette colonne sont rapportées de (Gao et al., 2015). Par ailleurs, nous indiquons (dans la colonne *BKS*) la valeur du meilleur résultat publié dans la littérature avant la parution de l'article de (Gao et al., 2015) – ce résultat avait presque toujours été atteint par Musliu, et assez souvent établi par lui. Une colonne indique la différence du minimum atteint par TS et par STS : cette colonne montre que la différence des valeurs *Min* est en faveur de STS pour 6 exemplaires, de TS dans 0 cas et égale dans les 74 cas restants : (+6, =74, -0). Le même calcul réalisé avec les moyennes donne le résultat suivant : (+17, =55, -8). Ceci montre que, dans nos tests, l'algorithme STS domine largement TS. Les résultats d'une comparaison entre notre meilleur algorithme STS et RWLS sont présentés dans le sous-chapitre 5.3.3.

En analysant les résultats, nous observons que, pour 16 jeux de données, des améliorations ont été apportées par STS sur les records antérieurs. Ces jeux se répartissent entre les catégories suivantes : 3 *CYC* (*CYC09*, *CYC10* et *CYC11*), 5/5 *NRG* (1-5), 3/5 *NRE* (*NRE2*, *NRE3* et *NRE5*), 2/5 *A* (*A1* et *A3*) et 3/5 *D* (*D2*, *D4* et *D5*). Dans la suite, nous appellerons ces 16 jeux de données les jeux difficiles.

5.3.3 Records obtenus avec l'algorithme STS

Le tableau 5.3 affiche les records obtenus avec l'algorithme STS sur les 16 jeux difficiles. Le tableau affiche pour chaque jeu de données :

- OBKS : l'ancien record obtenu avant 2015 ;
- RWLS : le meilleur résultat obtenu par l'algorithme RWLS, tel qu'il est rapporté dans (Gao et al., 2015) ;
- STS : le meilleur résultat obtenu par STS dans nos expériences systématiques ;
- OurRec : le meilleur résultat obtenu avec STS, incluant d'autres tests réalisés durant d'autres expériences, parfois avec un paramétrage différent.

Le tableau ne comporte que les 16 jeux que nous qualifions de difficiles. Pour tous les autres jeux de données, RWLS et STS atteignent l'ancien record sans l'améliorer.

Tableau 5.3 Tableau des nouveaux records établis avec l'algorithme STS

Exemplaire	OBKS	RWLS	STS	OurRec	RWLS -OBKS	STS -OBKS	OurRec -OBKS	STS -RWLS	OurRec -RWLS
CYC09	774	772	772	772	-2	-2	-2	0	0
CYC10	1792	1798	1792	1792	6	0	0	-6	-6
CYC11	4088	3968	3968	3968	-120	-120	-120	0	0
NRG1	61	61	60	60	0	-1	-1	-1	-1
NRG2	62	61	60	60	-1	-2	-2	-1	-1
NRG3	62	61	61	60	-1	-1	-2	0	-1
NRG4	62	61	61	60	-1	-1	-2	0	-1
NRG5	62	61	61	60	-1	-1	-2	0	-1
NRE2	17	16	16	16	-1	-1	-1	0	0
NRE3	17	16	16	16	-1	-1	-1	0	0
NRE5	17	16	16	16	-1	-1	-1	0	0
A1	39	38	38	38	-1	-1	-1	0	0
A3	39	38	38	38	-1	-1	-1	0	0
D2	25	24	24	24	-1	-1	-1	0	0
D4	25	24	24	24	-1	-1	-1	0	0
D5	25	24	24	24	-1	-1	-1	0	0
Bilan					14	15	15	3	6

Les cinq colonnes suivantes indiquent des différences permettant des comparaisons entre certaines paires de résultats. La dernière ligne "Bilan" correspond aux nombres de records obtenus par chaque algorithme. RWLS obtient tous les records antérieurs (OBKS) et 14 nouveaux records, mais fait moins bien une fois. STS obtient tous les records antérieurs et 15 nouveaux records ; il ne fait jamais moins bien que l'ancien record. STS réalise tous les résultats de RWLS + 3 améliorations : 2 améliorations sur des jeux déjà améliorés par RWLS + 1 pour un nouveau jeu. Nos meilleurs résultats obtenus durant d'autres séries de tests avec différents paramètres nous ont permis de battre 3 records de plus.

5.3.4 Comparaison entre les algorithmes STS et RWLS

Nous avons réalisé une série de tests pour comparer l'efficacité de notre algorithme STS à celui du meilleur algorithme de la littérature : l'algorithme RWLS. Pour ces nouveaux tests, les deux algorithmes disposeront du même temps CPU par exécution.

Pour notre algorithme, nous utilisons la meilleure implémentation possible selon le jeu de données. Les auteurs de (Gao et al., 2015) nous ont fourni leur code et nous avons compilé ce code sur notre machine. Les deux algorithmes (STS et RWLS) ont été compilés avec la même option de compilation (l'option O2).

Nous avons restreint ces tests aux 16 jeux dits difficiles. Nous avons réalisé deux expériences : la première avec un temps CPU fixé à 60 secondes, et la deuxième en 600 secondes. Chaque algorithme est exécuté 10 fois sur chaque jeu de données.

Le tableau 5.4 affiche les résultats obtenus en 60 secondes. Pour chacun des deux algorithmes STS et RWLS, le tableau affiche les deux mesures de performances suivantes : le meilleur résultat (le meilleur recouvrement) obtenu (*Min*) et la moyenne des résultats obtenus (*Avg*) sur les 10 exécutions. Une colonne STS-RWLS est associée à chacune des deux mesures qui donne l'information sur la différence de performance entre notre algorithme STS et l'algorithme RWLS. Une valeur négative dans l'une de ces deux colonnes indique une supériorité de notre algorithme alors qu'une valeur positive indique une supériorité de l'algorithme RWLS. Les 3 dernières lignes "av.STS", "av.RWLS" et "=" correspondent respectivement au nombre de fois que STS fait mieux, au nombre de fois que RWLS fait mieux et au nombre de fois que les 2 algorithmes sont à égalité. En analysant le tableau 5.4, nous constatons que RWLS fait légèrement mieux que STS en termes de *Min* – RWLS obtient 3 records alors que STS en obtient seulement 2. Sur les moyennes, RWLS garde sa supériorité en faisant mieux sur 8 exemplaires contre seulement 5 pour STS.

Le tableau 5.5 affiche les résultats obtenus en 600 secondes. Dans cette expérience, nous observons que STS domine en termes de records – STS obtient 2 records contre un seul record obtenu par RWLS. Sur les moyennes, RWLS fait toujours légèrement mieux que STS – RWLS obtient de meilleures moyennes sur 7 exemplaires contre seulement 5 pour STS.

Tableau 5.4 Comparaison entre STS et RWLS (CPU=60 secondes)

Exemplaire	Min			Avg		
	STS	RWLS	STS-RWLS	STS	RWLS	STS-RWLS
CYC09	772	772	0	773.2	776.5	-3.3
CYC10	1796	1800	-4	1800.5	1802.6	-2.1
CYC11	4035	4229	-194	4112.7	4245.4	-132.7
NRG1	61	61	0	61.5	61.4	+0.1
NRG2	61	61	0	61.7	61.9	-0.2
NRG3	62	61	+1	62.4	61.8	+0.6
NRG4	61	61	0	62.1	62	+0.1
NRG5	62	62	0	62.3	62.2	+0.1
A1	38	38	0	38.7	38.6	+0.1
A3	38	38	0	38.7	38.5	+0.2
NRE2	17	16	+1	17	16.9	+0.1
NRE3	17	17	0	17	17	0
NRE5	17	17	0	17	17	0
D2	25	24	+1	25	24.6	+0.4
D4	24	24	0	24.8	24.9	-0.1
D5	25	25	0	25	25	0
av. STS			2			5
av. RWLS			3			8
=			11			3

Tableau 5.5 Comparaison entre STS et RWLS (CPU=600 secondes)

Exemplaire	Min			Avg		
	STS	RWLS	STS-RWLS	STS	RWLS	STS-RWLS
CYC09	772	772	0	772.8	773.3	-0.5
CYC10	1792	1798	-6	1798.8	1799.4	-0.6
CYC11	4021	3994	+27	4063.1	4033.7	+29.4
NRG1	60	60	0	60.5	60.7	-0.2
NRG2	60	60	0	60.8	60.9	-0.1
NRG3	61	61	0	61.5	61	+0.5
NRG4	61	61	0	61.4	61.2	+0.2
NRG5	61	61	0	61.2	61	+0.2
A1	38	38	0	38	38	0
A3	38	38	0	38	38	0
NRE2	16	16	0	16.9	16.9	0
NRE3	17	17	0	17	17	0
NRE5	16	17	-1	16.8	17	-0.2
D2	24	24	0	24.8	24	+0.8
D4	24	24	0	24.6	24.1	+0.5
D5	24	24	0	24.8	24.3	+0.5
av. STS			2			5
av. RWLS			1			7
=			13			4

5.4 Expériences réalisées avec la liste taboue réactive

Dans ce sous-chapitre, nous présentons les résultats des expériences réalisées avec l'algorithme RTS. L'objectif de ces tests est d'analyser le comportement de l'algorithme RTS.

Dans la première expérience, nous testons l'algorithme RTS dans sa version décision (kTarget est fixe) sur l'exemplaire *CYC08* avec $kTarget = 342$. La valeur du paramètre *period* est fixée à 100. Nous fixons la valeur *seuil* à 2, sachant que la valeur initiale de la variable *cyclage* est 1– ce qui correspond à zéro collision.

La figure 5.11 affiche l'évolution des valeurs des variables *cyclage* et *tt* en fonction du nombre d'itérations avec l'exemplaire *CYC08*. Sur cette figure, l'axe vertical principal (à gauche) correspond aux valeurs de la variable *tt* alors que l'axe vertical secondaire correspond aux valeurs de la variable *cyclage*. Nous précisons que la figure correspond à une seule exécution.

Sur cette figure, nous observons que la valeur *tt* diminue progressivement sans produire des cycles jusqu'à atteindre la valeur 7, ce qui correspond à une perte de 80 % de la valeur initiale, à l'itération 5400. À cet instant, la valeur *cyclage* augmente brusquement et dépasse la valeur *seuil*. L'algorithme agit directement sur la valeur de la variable *tt* en l'augmentant. Après cette dernière augmentation, l'algorithme cesse de produire des cycles. Nous pouvons faire les mêmes observations tout au long de l'axe horizontal. À chaque fois la valeur *tt* atteint une valeur dans l'intervalle [4,10], la valeur *cyclage* dépasse la valeur *seuil* et l'algorithme agit en augmentant la valeur *tt* et cesse de produire des cycles. Nous pouvons conclure que notre mécanisme fonctionne bien sur cet exemplaire, mais qu'il détecte les cycles à des valeurs de *tt* très inférieures à la valeur idéale pour cet exemplaire ($tt^* = 22$) qui permet d'obtenir de bons résultats.

Dans la deuxième expérience, nous testons l'algorithme RTS dans sa version décision sur un des plus gros exemplaires de la ORLIB : *CYC10* avec une valeur $kTarget = 1820$. Nous utilisons les mêmes valeurs des paramètres de la première expérience.

La figure 5.12 affiche les résultats obtenus avec l'exemplaire *CYC10*. Sur cette figure, l'axe vertical principal correspond aux valeurs de la variable *tt* alors que l'axe vertical secondaire correspond aux valeurs de la variable *cyclage*. Nous précisons que la figure correspond à une seule exécution.

Nous observons sur cette figure que la valeur *tt* diminue progressivement jusqu'à atteindre une valeur nulle, mais l'algorithme ne produit jamais de cycles.

Sur cet exemplaire, l'algorithme RTS ne réussit pas à détecter les cycles, car simplement l'algorithme stagne sans cycler. Ce constat peut s'expliquer par le fait que la recherche se

bloque dans des vallées profondes.

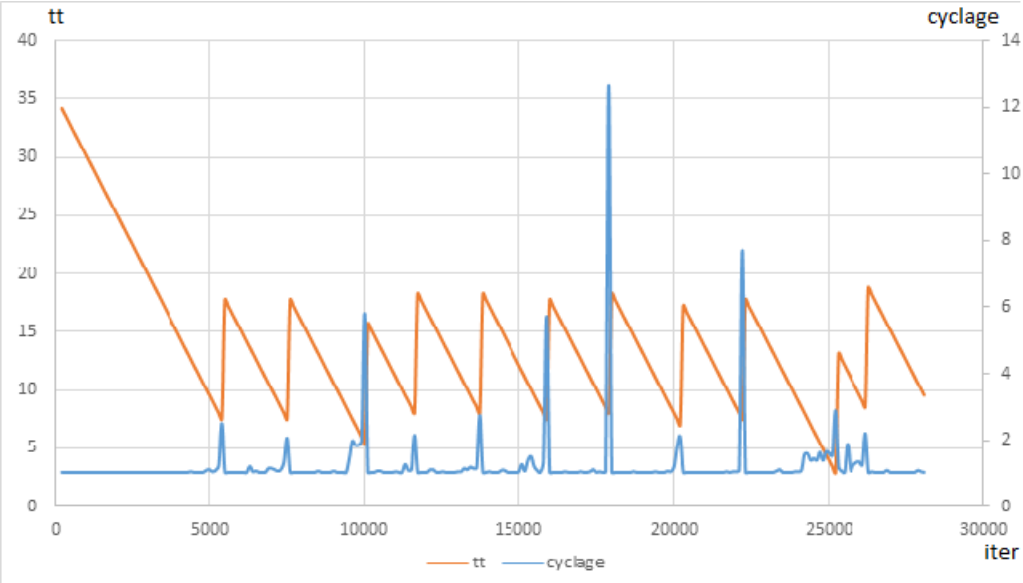

Figure 5.11 Évolution des valeurs des variables *cyclage* et *tt* en fonction du nombre d'itérations avec l'exemplaire *CYC08*

5.5 Tests portant sur l'implémentation

L'implémentation de nos algorithmes TS et STS a été présentée au chapitre 4. En réalité, nous avons développé deux implémentations différentes nommées A et A+ et l'algorithme peut utiliser l'une ou l'autre de ces implémentations. L'implémentation A+ est l'implémentation décrite au chapitre 4 ; cette implémentation utilise des files de priorité. L'implémentation A n'utilise pas les files de priorité, mais est identique à A+ sur les autres points.

Lors d'une itération de recherche locale, l'algorithme (TS out STS) fait appel aux trois procédures suivantes :

– les procédures *findIns*() et *findRem*() permenttent de sélectionner un mouvement ;
– la procédure *update*() met à jour les structures de données.

Les procédures *findIns*() et *findRem*() varient selon que l'algorithme est TS ou STS : dans TS, le meilleur mouvement est sélectionné, alors que dans STS un choix biaisé est réalisé. Dans l'implémentation de base A, ces procédures sont implantées en parcourant le tableau *Delta*[ ] en temps linéaire par rapport à la taille n du tableau. Dans l'implémentation A+, cette tâche est assurée par les files de priorité. La procédure *update*() met à jour les tableaux *Cov*[ ] et *Delta*[ ] et, avec l'implémentation A+, réalise des mises à jour dans les files de priorité. On peut donc attendre deux effets de l'utilisation des files de priorité :

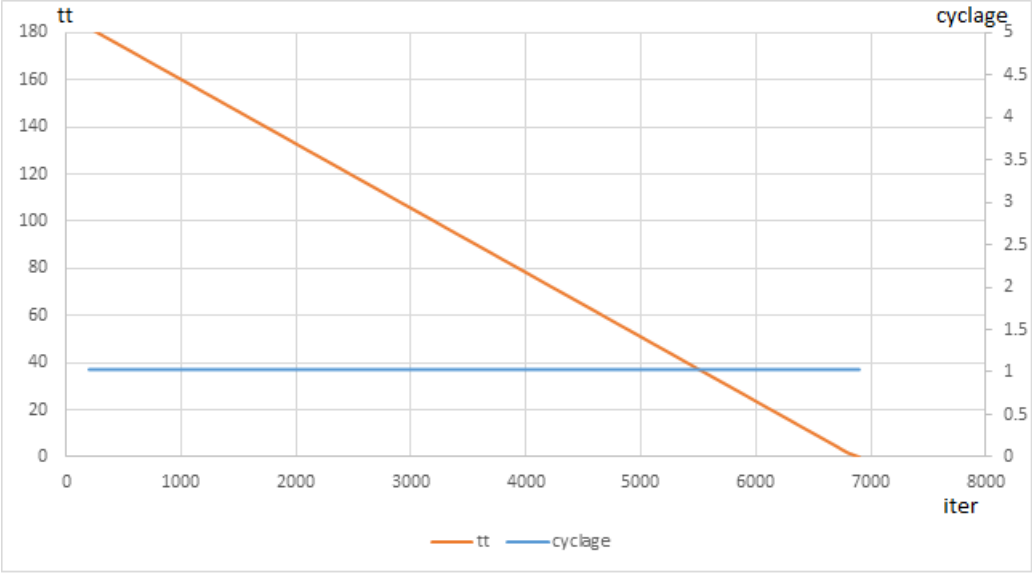

Figure 5.12 Évolution des valeurs des variables *cyclage* et *tt* en fonction du nombre d'itérations avec l'exemplaire *CYC10*

– les mises à jour sont plus lentes, car, lorsque le score d'un mouvement change, la variable correspondante est déplacée d'un Bucket à un autre ;
– la recherche du meilleur mouvement est généralement beaucoup plus rapide, car on évite de visiter un par un chacun des éléments du tableau *Delta*[ ].

5.5.1 Influence des listes de priorité sur le temps de calcul

Le tableau 5.6 présente des informations recueillies sur une sélection comprenant 7 jeux de données. Pour chacun d'entre eux, le tableau indique d'abord le nom de cet exemplaire ainsi que ses caractéristiques : les valeurs de n (nombre de sous-ensembles), m (nombre d'éléments) et la densité d. Le tableau affiche ensuite dans les colonnes suivantes le temps CPU (en secondes) utilisé pour réaliser un million d'itérations avec l'algorithme TS avec chacune des deux implémentations A et A+, ainsi que le ratio cpu(TS,A+)/cpu(TS,A). Les mêmes informations sont ensuite fournies pour l'algorithme STS.

Si nous observons le ratio entre le temps de calcul de TS avec les implémentations A+ et A, nous observons que l'implémentation A+ est plus rapide que A pour tous les jeux, sauf les plus denses : le taux d'accélération va jusqu'à 14 (STS, jeux *CYC11*). Par contre l'implémentation A+ est moins rapide que A pour les jeux *NRE*. Pour le jeu *NRG1*, les deux implémentations ont des temps comparables. Des observations analogues peuvent être réalisées au sujet de l'algorithme STS.

Finalement, si nous observons le ratio des temps de calcul entre TS et STS pour une même implémentation (A ou A+), nous observons que TS est toujours un peu plus rapide, avec un écart généralement compris entre 0% et 20%.

Tableau 5.6 Temps d'exécution des deux implémentations A et A+ pour les algorithmes TS et STS

Exemplaire	n	m	d	TS			STS		
				Impl.A	Impl.A+	Acc.	Impl.A	Impl.A+	Acc.
CYC08	1024	1790	0.4	3.21	0.79	**4.06**	3.95	0.93	**4.25**
CYC09	2300	4600	0.2	6.80	1.14	**5.96**	8.78	1.26	**6.97**
CYC10	5100	11500	0.08	16.07	2.05	**7.84**	24.21	2.41	**10.05**
CYC11	11264	28160	0.02	39.79	4.07	**9.78**	60.84	4.28	**14.21**
A1	3000	300	2	4.28	3.23	**1.33**	7.07	3.46	**2.04**
NRG1	10000	1000	5	14.54	15.34	0.95	23.55	15.33	**1.54**
NRE1	5000	500	10	29.42	93	0.32	33.01	87.33	0.38

5.5.2 Décomposition du temps de calcul entre les différentes procédures

Les figures 5.13 et 5.14 se réfèrent aux mêmes exécutions que la figure précédente, mais fournissent une information plus détaillée. En effet, chacun de ces temps de calcul est maintenant décomposé entre les temps de calcul de chacune des trois procédures *findIns*(), *findRem*() et *update*() décrites plus haut. Pour obtenir ces résultats, nous avons utilisé l'outil de profilage proposé dans l'environnement de programmation Visual Studio.

Ces deux figures 5.13 et 5.14 confirment les résultats obtenus dans le tableau précédent. Nous pouvons constater que les files de priorité permettent, sur les exemplaires peu denses (*CYC*), de réduire le temps CPU des procédures *findIns*() et *findRem*() avec un surcoût négligeable sur la procédure *update*(). Sur les exemplaires denses (*NRE*), l'utilisation des files de priorité multiplie par 3 le temps CPU de la procédure *update*() sans obtenir un gain remarquable sur les procédures *findIns*() et *findRem*(). Pour le jeu *NRG1*, l'accélération obtenue sur les procédures *findIns*() et *findRem*() grâce aux files de priorité est équivalente au surcoût de leur utilisation sur la procédure *update*().

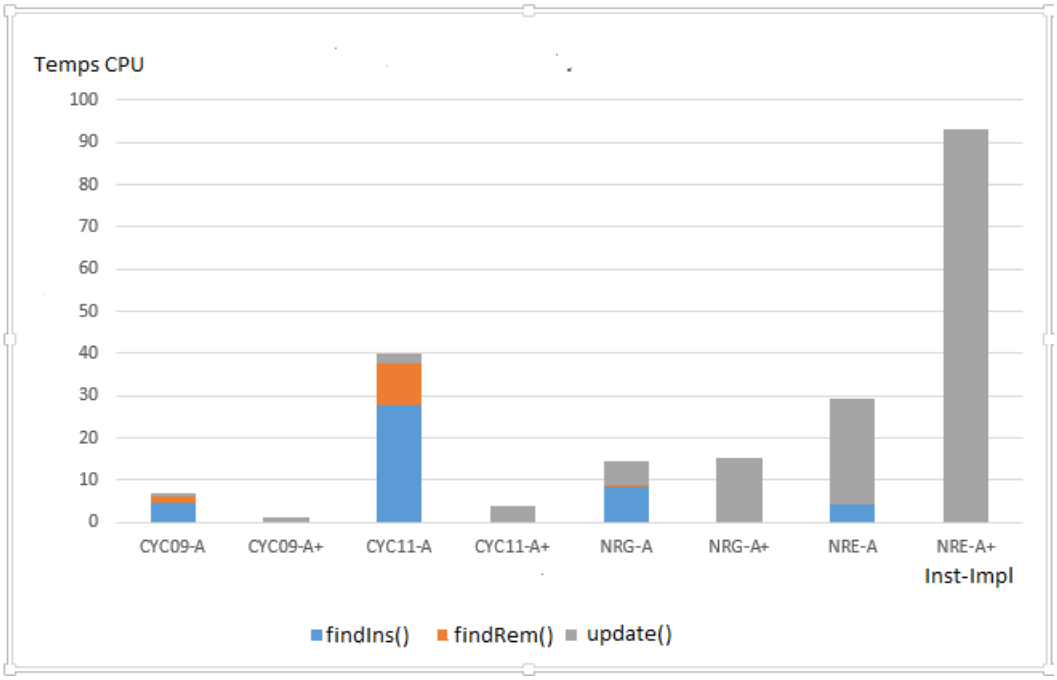

Figure 5.13 Algorithme TS : Temps CPU total en fonction de l'exemplaire et de l'implémentation utilisée et répartition du temps entre les différentes fonctions

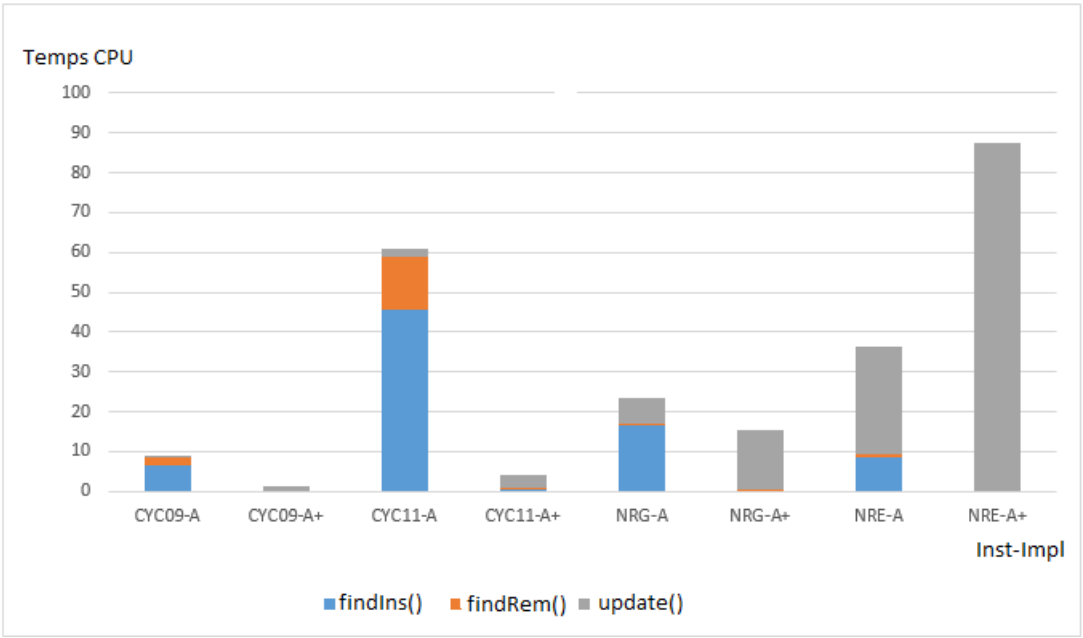

Figure 5.14 Algorithme STS : Temps CPU total en fonction de l'exemplaire et de l'implémentation utilisée et répartition du temps entre les différentes fonctions

CHAPITRE 6 CONCLUSION ET RECOMMANDATIONS

6.1 Synthèse des travaux réalisés

Dans ce mémoire, nous avons proposé de nouvelles heuristiques pour le problème USCP.

Notre point de départ était un algorithme tabou analogue à celui proposé dans la littérature. Nous avons poursuivi deux objectifs généraux : mettre en œuvre une technique pour régler efficacement les paramètres et développer une implémentation efficace.

Réglage des paramètres de l'algorithme tabou Notre premier objectif consistait à trouver une manière efficace de régler les paramètres de l'algorithme tabou. Pour réaliser cet objectif, nous avons exploré plusieurs approches.

La première approche explorée consiste simplement à trouver deux formules générales donnant la valeur des deux paramètres en fonction des paramètres du jeu de donnée et de la taille de la solution. Nous n'avons pas pu obtenir une formule simple (combinaison linéaire), mais nous avons effectivement obtenu une formule un peu plus compliquée et apparemment satisfaisante. Les résultats des tests réalisés avec l'algorithme tabou en utilisant ces formules sont sensiblement meilleurs que les résultats obtenus par (Musliu, 2006).

La deuxième approche explorée est l'utilisation d'une technique classique proposée dans la littérature : la méthode de liste tabou réactive proposée par (Battiti and Tecchiolli, 1994). Nous avons adapté la méthode de liste tabou réactive au problème USCP et testé cette technique sur des jeux de taille différente. Pour les jeux de petite taille, l'algorithme détecte en effet des cycles lorsque le paramètre tabou devient très petit, ce qui permet d'augmenter la valeur de celui-ci. Pour ces jeux, nous avons observé que, comme on pouvait le craindre, le réglage obtenu avec la liste tabou réactive conduit à utiliser une liste beaucoup plus courte que la valeur souhaitable. Mais ceci n'est pas le plus grave. Pour les jeux de données de grande taille (comme le *CYC10*), nous avons observé que, avec une liste taboue de longueur 0, l'algorithme stagne gravement, mais ne produit pas de cycles. Ceci rend la technique de liste taboue réactive totalement inopérante. En définitive, cette technique ne semble pas pouvoir être utilisée avec profit dans notre algorithme tabou.

Combiner l'algorithme tabou avec le recuit simulé Nous avons implanté un algorithme hybride qui combine l'algorithme tabou avec le recuit simulé (l'algorithme SATS). Nous avons réalisé une étude expérimentale qui nous a permis d'observer et d'analyser de

manière détaillée le comportement de l'algorithme SATS avec différents réglages, dans le cas d'une température fixe ou d'une température qui décroît. Nos tests avec l'algorithme SATS avec température décroissante ont révélé que la performance de l'algorithme se dégrade considérablement lorsque la température initiale n'est pas suffisamment basse. Il s'agit d'un phénomène inattendu et qui, à notre connaissance, n'a pas été décrit dans la littérature. Ceci indique que l'algorithme SATS avec décroissance de la température n'est pas adapté pour traiter efficacement le problème USCP.

Finalement, nous avons développé et testé un algorithme tabou stochastique (STS), qui ressemble à l'algorithme SATS, à la différence que la température ne décroît pas à partir d'une valeur élevée, mais oscille simplement entre des valeurs que nous prenons la précaution de choisir très petites. Pour notre algorithme STS, nous avons pu obtenir une formule simple pour le réglage des listes taboues (une fonction linéaire qui dépend seulement de la taille de la solution).

Performances de notre algorithme tabou stochastique (STS) Nous avons réalisé de tests étendus pour évaluer l'efficacité de l'algorithme STS, en utilisant le réglage trouvé. Les résultats sont encore meilleurs que ceux obtenus avec notre algorithme TS. Les résultats de STS sont même globalement aussi bons que ceux obtenus par l'algorithme RWLS de (Gao et al., 2015) publié très récemment. L'algorithme STS fait mieux que l'algorithme de Musliu sur 15 jeux de données et atteint le meilleur résultat sur tous les autres jeux. De plus, notre algorithme STS a permis de battre l'algorithme RWLS sur 6 jeux de données et d'atteindre son meilleur résultat sur tous les autres jeux. Nous avons aussi comparé les algorithmes STS et RWLS à temps CPU égal. Nous n'avons pas constaté la supériorité d'un algorithme sur l'autre.

Le développement de l'implémentation Notre deuxième objectif général consistait à développer une implémentation efficace. L'implémentation que nous avons développée présente deux caractéristiques. La première est l'utilisation d'une technique totalement incrémentale pour la mise à jour des scores des mouvements applicables à la configuration courante. La deuxième caractéristique de notre implémentation est l'utilisation de files de priorité qui accélèrent la sélection d'un mouvement lors d'une itération.

Efficacité de l'implémentation Nous avons testé l'impact de l'utilisation des files de priorité sur la rapidité de l'algorithme et observé le temps utilisé par différentes procédures de bas niveau. Nous avons pu ainsi quantifier le ralentissement des mises à jour et l'accélération de la sélection d'un mouvement, ceci en fonction de la densité de l'exemplaire traité. Pour

ce qui est des jeux de données standard de la littérature, les files de priorité permettent de réduire le temps de calcul d'un facteur atteignant 5 sur les jeux les moins denses. Les files de priorité ralentissent le temps de calcul pour une seule famille de jeux de données, celle qui contient les jeux les plus denses (la famille *NRE*). Pour les jeux denses, il nous suffit d'utiliser la version de base de l'implémentation qui n'utilise pas de files de priorité.

6.2 Limitations des travaux réalisés et améliorations futures

On peut imaginer des améliorations possibles de notre algorithme STS. Pour ce qui est de l'implémentation, on observe que les files de priorité ralentissent sensiblement les mises à jour. On pourrait imaginer de modifier leur implémentation en les faisant fonctionner de manière plus paresseuse, en évitant certaines mises à jour. Par exemple, une version élémentaire de cette idée consisterait à éviter de mettre à jour les mouvements dont le score est supérieur à un certain seuil.

On pourrait aussi explorer d'autres techniques pour régler les paramètres de liste taboue. Par exemple, il serait possible de faire varier systématiquement la valeur du paramètre tabou en lui faisant adopter périodiquement de petites valeurs (pour intensifier la recherche) et de grandes valeurs (pour diversifier la recherche). Une telle technique a été proposée par (Galinier and Boujbel, 2011).

Nos travaux ont été réalisés en parallèle avec ceux de (Gao et al., 2015) et en nous basant sur la même idée générale de greffer des améliorations sur l'algorithme proposé initialement par (Musliu, 2006). Ces auteurs ont développé un algorithme nommé RWLS. En analysant cet algorithme, on peut observer que les modifications apportées à l'algorithme initial proposées par (Gao et al., 2015) se sont orientées vers des voies différentes des nôtres. Une des techniques introduites par ces auteurs, et qui semble particulièrement intéressante, est l'utilisation de pondérations auto-adaptatives dans la fonction de coût. Une autre innovation proposée par ces auteurs consiste à réduire le nombre de mouvements explorés à chaque itération. Il serait intéressant de tester et d'évaluer chacune des caractéristiques de l'algorithme RWLS. On pourrait d'ailleurs envisager de combiner certaines de ces caractéristiques avec celles que nous avons introduites dans notre algorithme STS.

RÉFÉRENCES

E. Balas, “A class of location, distribution and scheduling problems : Modeling and solution methods”, *Carnegie Mellon University.Design Research Center*, 1982.

E. Balas et A. Ho, “Set covering algorithms using cutting planes, heuristics,and subgradient optimization : a computational study”, *Mathematical Programming*, vol. 60, pp. 12–37, 1980.

R. Battii et M. Protasi, “Reactive local search for the maximum clique problem”, *Algorithmica*, vol. 29, p. 610–637, 2001.

R. Battiti et G. Tecchiolli, “The reactive tabu search”, *Informs Journal on Computing*, pp. 126–140, 1994.

J. Bautista et J. Pereira, “A grasp algorithm to solve the unicost set covering problem”, *Computers and Operations Research*, vol. 34, pp. 3162–3173, 2007.

J. Beasley, “An algorithm for set covering problems”, *European Journal of Operational Research*, vol. 93, pp. 31–85, 1987.

——, “Or-library : Distributing test problems by electronic mail”, *Journal of the Operational Research Society*, vol. 41(11), p. 1069–1072, 1990.

——, “A lagrangian heuristic for set covering problems”, *Naval Research Logistics*, vol. 37, pp. 151–164, 1990.

N. Bilal, “Métaheuristiques hybrides pour les problèmes de recouvrement et recouvrement partiel d’ensembles appliquées au problème de positionnement des trous de forage dans les mines”, Thèse de doctorat, École Polytechnique de Montréal, 2014.

N. Bilal, P. Galinier, et F. Guibault, “A new formulation of the set covering problem for metaheuristic approaches”, *Operations Research*, vol. 2013, 2013.

S. Birbil et S. Fang, “An electromagnetism-like mechanism for global optimization”, *Journal of Global Optimization*, vol. 25, p. 263–282, 2003.

H. L. Bodlaender, “Discovering treewidth”, *31st Conference on Current Trends in Theory and Practice of Computer Science*, vol. 3381, pp. 1–16, 2005.

D. Brélaz, “New methods to color the vertices of a graph”, *Communications of the ACM*, vol. 22, p. 251–256, 1979.

A. Caprara, M. Fischetti, et P. Toth, “A heuristic method for the set covering problem”, *Operations Research*, vol. 47, pp. 730–743, 1999.

S. Ceria, P. Nobili, et A. Sassano, “Set covering problem”, *Annotated Bibliographies in Combinatorial Optimization. John Wiley and Sons, USA*, pp. 415–428, 1998.

V. Cerny, “Thermodynamical approach to the travelling salesman problem”, *J. Optimization Theory and Applications*, vol. 45, pp. 41–51, 1985.

M. Chams, A. Hertz, et D. de Werra, “Some experiments with simulated annealing for coloring graphs”, *European J. of Operational Research*, vol. 32, pp. 260–266, 1987.

V. Chvatal, “A greedy heuristic for the set-covering problem”, *Mathematics of Operations Research*, vol. 4, p. 233–235, 1979.

T. Feo et M. Resende, “Greedy randomized adaptive search procedures”, *J. Global Optim*, pp. 109–133, 1995.

——, “A probabilistic heuristic for a computationally difficult set covering problem”, *Operations Research Letters*, vol. 8, p. 67–71, 1989.

P. Galinier et C. Boujbel, Z. Micheal, “An efficient memetic algorithm for the graph partitioning problem”, *Annals of Operations Research*, vol. 191, p. 1–22, 2011.

C. Gao, X. Yao, T. Weise, et J. Li, “An efficient local search heuristic with row weighting for the unicost set covering problem”, *European Journal of Operational Research*, vol. 246, pp. 750–761, 2015.

M. Garey et D. Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, USA, 1979.

F. Glover, “Tabu search - part i”, *Informs Journal on Computing*, pp. 190–206, 1989.

——, “Tabu search - part ii”, *Informs Journal on Computing*, pp. 4–32, 1990.

F. Glover et M. Laguna, “Tabu search”, *Kluwer Academic Publishers, Boston, USA*, 1997.

D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing, 1989b.

——, *Genetic algorithms.* Addison-Wesley Longman Publishing, 1989a.

G. Gottlob, N. Leone, et F. Scarcello, "Comparison of structural csp decomposition methods", *Artificial Intelligence*, pp. 243–282, 2000.

J. Greene et K. Supowit, "Simulated annealing without rejected moves", *IEE Transactions On Computer-Aided Design*, vol. 5, pp. 221–228, 1986.

T. Grossman et A. Wool, "Computational experience with approximation algorithms for the set covering problem", *European Journal of Operational Research*, vol. 92, pp. 81–101, 1997.

J. Hao, P. Galinier, et M. Habib, "Méthaheuristiques pour l'optimisation combinatoire et l'affectation sous contraintes", *Revue d'Intelligence Artificielle*, 1999.

A. Hertz et D. de Werra, "Using tabu search for graph coloring", *Revue d'Intelligence ArtificielleComputing*, vol. 39, pp. 345–351, 1987.

J. Holland, *Adaptation in natural and artificial systems.* University of Michigan press, 1975.

S. D. G. Kirkpatrick et M. P. Vecchi, "Optimization by simulated annealing", *Science*, vol. 220, pp. 671–680, 1983.

G. Lan, G. DePuy, et G. Whitehouse, "An effective and simple heuristic for the set covering problem", *European Journal of Operational Research*, p. 1387–1403, 2007.

N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, et E. Teller, "Simulated annealing", *J. Chem. Phys*, pp. 1087–1092, 1953.

S. Minton, M. Johnston, A. Philips, et P. Laird, "Minimizing conflicts : a heuristic repair method for constraint satisfaction and scheduling problems", *Artificial Intelligence*, vol. 58, pp. 161–205, 1992.

N. Musliu, "Local search algorithm for unicost set covering problem", *Advances in Applied Artificial Intelligence - Lecture Notes in Computer Science*, vol. 4031, pp. 302–311, 2006.

Z. Naji-Azimi, P. Toth, et L. Galli, "An electromagnetism metaheuristic for the unicost set covering problem", *European Journal of Operational Research*, vol. 205, pp. 290–300, 2010.

S. Shauger, "Results on the erdős–gyárfás conjecture in $k_{1,m}$-free graphs", *Proc. 29th Southeastern Int. Conf. Combinatorics, Graph Theory, and Computing*, p. 61–65, 1998.

M. Yaghini, M. Sarmadi, et M. Momeni, “A local branching approach for the set covering problem”, *International Journal of Industrial Engineering And Production Research*, vol. 25, pp. 95–102, 2014.

ANNEXE A Résultats détaillés des algorithmes STS et TS

Tableau A.1 – Résultats détaillés des algorithmes STS et TS

Exemplaire	BKS	RWLS	STS						TS						Min	Avg
			Nbexe	Min	Avg	CpuT	Succ	IBest	Nbexe	Min	Avg	CpuT	Succ	IBest	STS-TS	STS-TS
41	38	38	20	38	38	64.38	20	20,738	20	38	38	55.24	20	22,296	0	0
42	37	37	20	37	37	68.14	20	5,018	20	37	37	61.56	20	3,833	0	0
43	38	38	20	38	38	81.16	20	4,751	20	38	38	57.22	20	3,502	0	0
44	38	38	20	38	38	65.94	20	186,416	20	38	38.10	55.66	18	192,626	0	-0.10
45	38	38	20	38	38	64.70	20	32,598	20	38	38	54.12	20	20,278	0	0
46	37	37	20	37	37	68.12	20	101,496	20	37	37.05	60.32	19	68,553	0	-0.05
47	38	38	20	38	38	65.32	20	67,963	20	38	38.05	53.76	19	69,186	0	-0.05
48	37	37	20	37	37	69.78	20	137,564	20	37	37	63.12	20	160,384	0	0
49	38	38	20	38	38	67.50	20	29,867	20	38	38	53.86	20	24,344	0	0
410	38	38	20	38	38	66.56	20	119,510	20	38	38	55.36	20	153,365	0	0
51	34	34	20	34	34	114.56	20	579,613	20	34	34	104.06	20	293,849	0	0
52	34	34	20	34	34	115.32	20	169,063	20	34	34	104.52	20	81,101	0	0
53	34	34	20	34	34	115.94	20	44,359	20	34	34	105.34	20	25,665	0	0
54	34	34	20	34	34	123.88	20	41,823	20	34	34	108.76	20	36,645	0	0
55	34	34	20	34	34	111.88	20	60,740	20	34	34	100.94	20	26,050	0	0
56	34	34	20	34	34	119.08	20	91,747	20	34	34	102.22	20	99,082	0	0
57	34	34	20	34	34	114.86	20	17,424	20	34	34	102.30	20	16,392	0	0
58	34	34	20	34	34	115.34	20	76,419	20	34	34	98.44	20	115,228	0	0
59	35	35	20	35	35	112.20	20	32,698	20	35	35	98.62	20	18,534	0	0
510	34	34	20	34	34	120.44	20	98,494	20	34	34	105.34	20	60,387	0	0
61	21	21	20	21	21	214.08	20	8,331	20	21	21	131.58	20	4,318	0	0
62	20	20	20	20	20	210.64	20	52,913	20	20	20	129.08	20	19,645	0	0
63	21	21	20	21	21	197.26	20	5,211	20	21	21	128.94	20	3,760	0	0
64	20	20	20	20	20	205.86	20	180,731	20	20	20	140.32	20	231,636	0	0
65	21	21	20	21	21	202.20	20	9,860	20	21	21	135.52	20	4,578	0	0
A1	39	38	20	38	38	198.82	20	22,810,000	20	38	38	177.82	20	20,049,300	0	0
A2	38	38	20	38	38	208.84	20	2,423,910	20	38	38	197.14	20	960,087	0	0
A3	39	38	20	38	38.05	189.56	19	12,897,400	20	38	38	167.46	20	17,676,800	0	0.05
A4	37	37	20	37	37	201.58	20	1,998,750	20	37	37	205.34	20	1,240,550	0	0
A5	38	38	20	38	38	194.66	20	215,695	20	38	38	171.90	20	104,788	0	0
B1	22	22	20	22	22	563.42	20	22,517	20	22	22	370.62	20	24,631	0	0
B2	22	22	20	22	22	561.50	20	21,072	20	22	22	379.38	20	22,015	0	0
B3	22	22	20	22	22	576.14	20	76,585	20	22	22	381.40	20	84,025	0	0
B4	22	22	20	22	22	603.28	20	107,510	20	22	22	398.24	20	135,200	0	0
B5	22	22	20	22	22	576.90	20	69,242	20	22	22	382.76	20	59,632	0	0
C1	43	43	20	43	43	286.20	20	129,600	20	43	43	244.96	20	159,210	0	0
C2	43	43	20	43	43	286.78	20	173,085	20	43	43	261.14	20	207,085	0	0
C3	43	43	20	43	43	303.74	20	104,738	20	43	43	277.86	20	404,738	0	0
C4	43	43	20	43	43	281.08	20	92,918	20	43	43	233.76	20	107,483	0	0
C5	43	43	20	43	43	285.98	20	349,557	20	43	43	264.80	20	472,357	0	0
D1	24	24	20	24	24	833	20	3,302,070	20	24	24	547.42	20	617,096	0	0
D2	25	24	20	24	24.50	806.68	10	35,166,200	20	24	24.35	552.84	13	25,734,800	0	0.15
D3	24	24	20	24	24.90	786.92	2	27,172,000	20	24	24.30	523.04	14	24,888,420	0	0.60
D4	25	24	20	24	24.65	824	7	26,438,600	20	24	24.25	553.08	15	27,157,300	0	0.40
D5	25	24	20	24	24.80	813	4	22,447,200	20	24	24.65	533.04	7	21,533,900	0	0.15

Suite page suivante

Tableau A.1 – *Suite ...*

			Nbexe	Min	Avg	CpuT	Succ	IBest	Nbexe	Min	Avg	CpuT	Succ	IBest	STS-TS	STS-TS
E1	5	5	20	5	5	202.34	20	506	20	5	5	138.76	20	305	0	0
E2	5	5	20	5	5	211.88	20	470	20	5	5	131.30	20	720	0	0
E3	5	5	20	5	5	201.10	20	845	20	5	5	124.38	20	685	0	0
E4	5	5	20	5	5	202.40	20	984	20	5	5	131.06	20	514	0	0
E5	5	5	20	5	5	199.40	20	772	20	5	5	125.64	20	381	0	0
NRE1	16	16	20	16	16.90	1910.22	2	47,773,200	20	16	17	1521.76	2	10,318,100	0	-0.10
NRE2	17	16	20	16	16.85	1938.56	3	28,217,000	20	17	17	1422.84	20	41,871	-1	-0.15
NRE3	17	16	20	16	16.85	1922.86	3	21,517,800	20	17	17.80	1487.50	4	9,751	-1	-0.95
NRE4	16	16	20	16	16.85	1894.92	3	45,404,800	20	17	17.20	1555.78	16	7,830	-1	-0.35
NRE5	17	16	20	16	16.85	1926.10	3	8,438,590	20	17	17.50	1325.72	10	53,418	-1	-0.65
NRF1	10	10	7	10	10	4981.70	7	517,004	10	10	10	3803.54	10	781,419	0	0
NRF2	10	10	7	10	10	4891.42	7	648,203	10	10	10	4118.80	10	708,473	0	0
NRF3	10	10	7	10	10	4955.20	7	400,581	10	10	10	3645	10	604,153	0	0
NRF4	10	10	7	10	10	5005.88	7	929,457	10	10	10	3523.42	10	1,246,201	0	0
NRF5	10	10	7	10	10	5035.74	7	1,033,714	10	10	10	4318.20	10	1,115,874	0	0
NRG1	61	61	20	60	60.50	923.82	10	29,485,300	20	60	60.50	957.96	10	29,068,500	0	0
NRG2	62	61	20	60	60.50	908.46	10	27,223,300	20	60	60.45	934.64	11	31,959,400	0	0.05
NRG3	62	61	20	61	61.05	889.70	19	19,402,500	20	61	61.10	941.22	18	21,842,600	0	-0.05
NRG4	62	61	20	61	61.05	916.42	19	14,554,900	20	61	61.10	951.44	18	24,581,000	0	-0.05
NRG5	62	61	20	61	61	909.56	20	14,097,100	20	61	61.10	992.78	18	25,469,100	0	-0.10
NRH1	34	34	15	34	34	2270.92	15	814,607	20	34	34.80	1720.40	6	1,068,490	0	-0.80
NRH2	34	34	15	34	34	2280.08	15	1,058,147	20	34	34.90	1680.94	5	1,309,310	0	-0.90
NRH3	34	34	15	34	34	2249.94	15	1,347,314	20	34	34.85	1698.94	4	928,674	0	-0.85
NRH4	34	34	15	34	34	2260.36	15	1,090,573	20	34	34.75	1735	6	1,733,350	0	-0.75
NRH5	34	34	15	34	34	2264.40	15	1,628,597	20	34	35.05	1755.36	4	1,946,520	0	-1.05
CLR10	25	25	20	25	25	443.80	20	924	20	25	25	296.47	20	814	0	0
CLR11	23	23	20	23	23	918.93	20	1,111	20	23	23	596.07	20	1,501	0	0
CLR12	23	23	20	23	23	2033.74	20	10,848	20	23	23	1460.73	20	12,952	0	0
CLR13	23	23	8	23	23	4712.60	8	1,066,385	16	23	23	2270.60	16	1,546,415	0	0
CYC06	60	60	20	60	60	126.07	20	1,317	20	60	60	102.20	20	3,015	0	0
CYC07	144	144	20	144	144	131.27	20	40,355	20	144	144	119.80	20	31,184	0	0
CYC08	342	342	20	342	342	133.33	20	181,791	20	342	342	129.40	20	1,519,610	0	0
CYC09	774	772	20	772	773.20	152.07	11	11,093,500	20	772	773.10	140.60	15	13,961,900	0	0.10
CYC10	1792	1798	20	1792	1799.50	171.27	1	98,205,200	20	1805	1808.85	155.27	1	92,651,600	-13	-9.35
CYC11	4088	3968	20	3968	4059.70	229.20	1	136,062,000	20	3969	4054.30	206.27	1	122,563,000	-1	5.40